UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
Programa de Pós-Graduação em Educação
Marcelo Bráulio Pedras

REDBLUE: *cluster* para pesquisa e ensino em Engenharia

Diamantina
2017

Marcelo Bráulio Pedras

REDBLUE: *cluster* para pesquisa e ensino em Engenharia

Dissertação apresentada ao Programa de Pós-Graduação em Educação como parte dos requisitos exigidos para obtenção do título de Mestre em Educação.

Orientador: Alexandre Ramos Fonseca
Coorientador: Euler Guimarães Horta

Diamantina
2017

Ficha Catalográfica – Serviço de Bibliotecas/UFVJM
Bibliotecário Anderson César de Oliveira Silva, CRB6 – 2618.

P371r Pedras, Marcelo Bráulio
RedBlue: cluster para pesquisa e ensino em Engenharia / Marcelo Bráulio Pedras. – Diamantina, 2017.
97 p. : il.

Orientador: Alexandre Ramos Fonseca
Coorientador: Euler Guimarães Horta

Dissertação (Mestrado Profissional – Programa de Pós-Graduação em Educação) - Universidade Federal dos Vales do Jequitinhonha e Mucuri.

1. Simulações computacionais. 2. Cluster de computadores. 3. Educação e pesquisa. 4. Programação paralela. 5. Programação distribuída. I. Fonseca, Alexandre Ramos. II. Horta, Euler Guimarães. III. Título. IV. Universidade Federal dos Vales do Jequitinhonha e Mucuri.

CDD 005

Elaborado com os dados fornecidos pelo(a) autor(a).

MARCELO BRÁULIO PEDRAS

RedBlue: cluster para pesquisa e ensino em Engenharia

Dissertação apresentada ao PROGRAMA DE PÓS-GRADUAÇÃO EM EDUCAÇÃO - STRICTO SENSU, nível de MESTRADO como parte dos requisitos para obtenção do título de MAGISTER SCIENTIAE EM EDUCAÇÃO

Orientador: Prof. Dr. Alexandre Ramos Fonseca

Coorientador: Prof. Dr. Euler Guimarães Horta

Data da aprovação : 13/11/2017

Prof.Dr. PAULO CESAR DE RESENDE ANDRADE - UFVJM

Prof.Dr. LIBARDO ANDRÉS GONZÁLEZ TORRES - UFVJM

Prof.Dr. EULER GUIMARAES HORTA - UFVJM

Prof.Dr. ALEXANDRE RAMOS FONSECA - UFVJM

DIAMANTINA

Dedico esta obra a minha esposa, Carolina, pela paciência e carinho sem os quais não seria possível concluir este programa de mestrado.

AGRADECIMENTOS

Agradeço ao Instituto de Ciência e Tecnologia, em especial ao diretor Lucas Franco Ferreira, por ceder os equipamentos de informática e rede, indispensáveis ao desenvolvimento deste trabalho. Aos meus amigos Anderson Matos Fernandes e Reinaldo Lívio Tameirão Duarte que sempre estiveram ao meu lado apesar de todas as adversidades. A meu orientador Alexandre Ramos Fonseca, o qual propôs este trabalho e a meu coorientador Euler Guimarães Horta, que certamente contribuiu para a melhoria da qualidade do texto desta obra.

RESUMO

Programas de computadores são muito utilizados para resolução de problemas complexos em engenharia. Atualmente, espera-se que um engenheiro saiba mais que apenas utilizá-los, sendo esta habilidade muito valorizada no mercado de trabalho. Tal habilidade possibilita que profissionais consigam utilizar um maior conjunto de ferramentas para solucionar problemas. As simulações computacionais, por exemplo, podem ser utilizadas como ferramenta de aquisição de conhecimento, permitindo que um profissional ou um estudante crie, teste e valide suas hipóteses. As simulações também são utilizadas em pesquisas científicas como alternativa a experimentos de difícil obtenção e na indústria para reduzir custos. Porém, uma simulação pode consumir mais recursos do que os disponíveis em um computador, tornando seu tempo de execução inviável. Uma forma barata de se obter mais desempenho é utilizando um *cluster* de computadores comuns. Dessa forma, seria possível utilizar os laboratórios de informática disponíveis para executá-las. Entretanto, isso implicaria em conhecimentos aprofundados em computação paralela e/ou distribuída por parte dos usuários, dificultado o desenvolvimento de aplicações. Com o objetivo de minimizar o tempo de execução de simulações complexas utilizando *clusters* e permitir que usuários com poucos conhecimentos em programação paralela e/ou distribuída possam utilizá-lo, este trabalho apresenta uma solução denominada "plataforma RedBlue". Essa plataforma recebe a aplicação do usuário e a executa nos nós do *cluster* de forma automática e transparente para o mesmo. Para testar a plataforma desenvolvida foram realizados testes com redes neurais artificiais e com um algoritmo genético simples, ambos buscando descobrir a melhor configuração de parâmetros para determinado problema. Utilizaram-se 60 máquinas de um laboratório de informática para testar a plataforma. Os resultados mostram que houve uma redução de até 98% no tempo de execução do experimento com redes neurais e 99,3% para o experimento com o algoritmo genético em comparação a execução sequencial. Esses resultados indicam que a plataforma é viável para utilização em laboratórios de informática, possibilitando uma redução considerável no tempo de execução de simulações complexas. A plataforma é aplicável a um número flexível de computadores, ajustando-se à capacidade dos laboratórios. Além disso, pode ser utilizada como instrumento útil ao ensino e pesquisa. Ressalta-se que a utilização de simulações computacionais para ensino e pesquisa contribui não apenas para a aprendizagem de conteúdos, mas também para o surgimento de habilidades necessárias ao mercado de trabalho do engenheiro.

Palavras-chave: Simulações computacionais. *Cluster* de computadores. Educação e pesquisa. Programação paralela. Programação distribuída.

ABSTRACT

Computer programs are commonly used to solve complex engineering problems, and it is expected from an engineer a more than hands-on experience in using these computer programs with the ability to develop them using a wide range of tools. Computational simulations, for instance, can be used as tools for knowledge acquisition allowing a professional or student to create, test and validate their hypotheses. Such simulations are used at an academic setting as an alternative to expensive experiments. However, a simulation can take more resources than those available in a single computer machine, rendering long execution times. To create a cluster of regular computers, such as the ones already available at computer labs, is a cheaper alternative to improve such execution times. One major drawback of this approach is that the user must be knowledgeable in parallel and distributed programming, which makes software development harder. To overcome such constraints, this work presents a solution named ”RedBlue platform”that receives and runs user's applications over a computer cluster in an automatic, transparent manner. To test the RedBlue platform, we performed a set of tests via artificial neural networks and a simplified genetic algorithm, whose main purpose was to search for the best-suited parameter configurations for the application problem at hand. To test the platform, the experiments were run using 60 computer machines from a computer lab. This study has identified a reduction in execution times of 98% for neural networks, and a reduction of 99,3% for the genetic algorithm, and also shown that the platform is suited for real-world applications of simulations at computer labs. Furthermore, the platform accepts a variable number of computers, easily adaptable to different academic environments, such as research and training. Lastly, we have noted that computational simulations not only contribute to research and learning, but also to develop the required industry skills.

Keywords: Computational simulation. Computer cluster. Education and research. Parallel programming. Distributed programming.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BIOS - *Basic Input/Output System*

DRb - *Distributed Ruby*

DRBL - *Diskless Remote Boot in Linux*

INPE - Instituto Nacional de Pesquisas Espaciais

IDE - *Integrated Development Environment*

IP - *Internet Protocol*

JSON - *JavaScript Object Notation*

LNCC - Laboratório Nacional de Computação Científica

MIT - *Massachusetts Institute of Technology*

MPI - *Message Passing Interface*

NFS - *Network File System*

PXE - *Preboot eXecution Environment*

RAM - *Random Access Memory*

RMI - *Remote Method Invocation*

Rb - Ruby

SSI - *Single System Image*

STDERR - *Standard Error*

STDIN - *Standard In*

STDOUT - *Standard Out*

UFVJM - Universidade Federal dos Vales do Jequitinhonha e Mucuri

VM - *Virtual Machine*

SUMÁRIO

1 INTRODUÇÃO

Uma prática comum nas engenharias é a utilização de programas de computador (*softwares*) para resolução de problemas complexos (TIWARI *et al.*, 2015). O *software* permite sistematizar e delimitar os pontos principais de um problema, tornando sua representação mais simples. O próprio exercício de modelagem contribui para uma melhor compreensão do problema. A identificação dos aspectos mais relevantes faz com que seja possível desmembrá-los, facilitando a análise aprofundada em pontos específicos. Por sua vez, o relacionamento entre as partes permite identificar o fluxo da informação e as dependências entre processos.

A capacidade de um engenheiro analisar o problema sob a perspectiva de um cientista da computação é uma característica muito valorizada no mercado de trabalho (NEUBERT *et al.*, 2015). Neste, destaca-se o setor industrial, uma vez que a identificação, compreensão e correção das ineficiências nos processos de produção podem gerar vantagens competitivas. Nesse contexto, uma das principais ferramentas utilizadas nas engenharias são as simulações computacionais.

Uma simulação consiste na construção de um modelo computacional simplificado pelo qual se pode aproximar um processo ou evento do mundo real. Esses modelos permitem a exploração de situações fictícias ou de alto risco, como: a manipulação de substâncias perigosas; experimentos muito caros ou de difícil obtenção; situações em que a periodicidade do evento torna a coleta de dados difícil, como o crescimento de árvores (muito longa) ou uma reação química (muito curta), entre outras (VALENTE, 1998). Na exploração da simulação o usuário pode alterar o comportamento do sistema a partir da manipulação de parâmetros e variáveis, tornando possível a análise do processo em diferentes cenários (MONTEIRO *et al.*, 2011).

Nas indústrias em que se desenvolvem novas tecnologias, como a automobilística, aviação, química, entre outras, a simulação computacional é amplamente utilizada para reduzir custos. Como exemplo, ao se projetar uma nova carroceria para um automóvel, deseja-se minimizar o arrasto aerodinâmico. Para isso, inúmeras simulações são realizadas em um computador antes de se construir um protótipo físico que será testado em um túnel de vento (WANG *et al.*, 2017).

Em resposta à necessidade do mercado de trabalho (SOUNDARAJAN, 1999), grandes universidades começaram a atualizar a metodologia de ensino e adequar o currículo para que o aluno desenvolva essas competências durante o curso (KOLAR; SABATINI, 1996; CROUCH; MAZUR, 2001; HALL *et al.*, 2002). Busca-se não somente transmitir conteúdos, mas ensinar ao aluno como estruturar seu raciocínio para, continuamente, buscar e absorver novos conhecimentos. Atualmente, essas práticas se encontram consolidadas em universidades como o MIT (2017), Harvard (2017) e Cambridge (2017).

A utilização de atividades práticas permite ao aprendiz construir e reconstruir seu esquema cognitivo baseado em suas próprias experiências (DALGARNO; KENNEDY; BENNETT, 2014). O aluno é submetido a problemas para os quais a solução não depende apenas da aplicação direta de um conhecimento adquirido em aulas expositivas, mas da capacidade de relacionar seus conhecimentos prévios aos recém adquiridos e formular hipóteses. Nesse

contexto, a simulação computacional serve como meio para que o aprendiz teste a validade das suas hipóteses, possibilitando que suas novas descobertas atualizem seu conhecimento prévio (JONG, 2006; CARDOSO; DICKMAN, 2012). Segundo Martin e Slater (2012), como resultado desses métodos de ensino, são desenvolvidas habilidades em comunicação, trabalho em equipe e resolução de problemas que permitem ao aluno identificar quando e porque aplicar conceitos teóricos com mais facilidade. Como consequência, forma-se um profissional com maior potencial para responder positivamente a situações desafiadores, enquadrando-se ao perfil desejado pelo mercado (LATTUCA; TERENZINI; VOLKWEIN, 2006).

Antes de ser aplicada na indústria e educação, a simulação computacional já vinha sendo amplamente utilizada para realização de pesquisa científica, principalmente como alternativa a experimentos onerosos ou de difícil obtenção (GRECA; SEOANE; ARRIASSECQ, 2014). Normalmente essas simulações envolvem uma série de cálculos complexos que, muitas vezes, possuem elevado custo computacional. Apesar do desenvolvimento expressivo dos computadores pessoais, seus recursos podem ser insuficientes para execução de simulações complexas em tempo aceitável.

Uma solução para esse problema seria a utilização de equipamentos mais robustos, como supercomputadores. No entanto, além do elevado custo de aquisição, esse tipo de equipamento exige um alto investimento em infraestrutura, uma vez que necessitam de um controle rigoroso de temperatura e alimentação elétrica dedicada (GHOLKAR; MUELLER; ROUNTREE, 2016). Além do investimento inicial, seu alto custo de operação, principalmente pelo elevado consumo de energia elétrica, e sua vida útil de apenas 5 anos (GHOLKAR; MUELLER; ROUNTREE, 2016) o torna uma solução restrita a grandes centros de pesquisa. Recentemente noticiou-se a dificuldade do INPE (Instituto Nacional de Pesquisas Espaciais) em manter seu supercomputador, Tupã, em operação devido ao alto custo da energia elétrica e problemas com renovação do contrato de garantia. Isso ocorreu devido aos cortes de investimento por parte do governo federal brasileiro (Bom dia Brasil, 2017). Algo similar aconteceu no LNCC (Laboratório Nacional de Computação Científica), que restringiu o acesso ao supercomputador Santos Dummont para diminuir gastos, visto que somente a conta de energia elétrica, cerca de quinhentos mil reais, consumia 80% do orçamento destinado ao LNCC (Folha de São Paulo, 2016).

Atualmente, de acordo com a lista elaborada pela TOP500 Team (2017), cerca de 86,4% dos 500 supercomputadores mais poderosos do mundo são implementados sob a forma de *clusters*. Um *cluster* de computadores é um sistema distribuído formado por um conjunto de computadores interligados por uma rede de alta velocidade com objetivo de processar tarefas de modo colaborativo. Embora os grandes centros de pesquisa utilizem servidores especialmente construídos para fins científicos, é possível implementar um *cluster* utilizando computadores comuns (QUINN, 2004; DATTI; UMAR; GALADANCI, 2015; SETIAWAN; MURDYANTORO, 2016).

No entanto, *clusters* possuem peculiaridades quanto à estrutura dos programas executados devido ao ambiente distribuído em vários computadores, necessitando de conhecimentos mais aprofundados em programação paralela e distribuída. Normalmente utiliza-se um sistema operacional de uso geral, como alguma distribuição Linux, em conjunto com uma interface de programação paralela, comumente MPI (*Message Passing Interface*), para comunicação e sincronização entre aplicações remotas.

Uma alternativa à programação distribuída seria a utilização de um sistema que ocultasse a natureza distribuída do *cluster*, apresentando-se ao usuário como um sistema unificado composto por recursos de múltiplos computadores. Um sistema de imagem única (SSI - *Single System Image*) poderia ser utilizado para esta finalidade. Como exemplos de *softwares* livres desse tipo, têm-se: OpenMosix, OpenSSI e Kerrighed (LOTTIAUX *et al.*, 2005). No entanto, todos esses foram desenvolvidos utilizando-se modificações de rotinas de baixo nível no *kernel* do sistema operacional, tornando-os dependentes das versões do *kernel* em que foram implementados. Como consequência, esses projetos não dão suporte aos sistemas operacionais atuais, que utilizam versões mais novas do *kernel*.

Em instituições de ensino e pesquisa, é comum que existam laboratórios de informática com várias máquinas interligadas em rede e que podem ficar ociosos em parte do tempo. Dessa forma, essas instituições poderiam utilizar seus laboratórios de informática como *clusters* para execução de simulações computacionais.

Com o objetivo de facilitar o uso de simulações computacionais utilizando máquinas comuns, este trabalho apresenta uma nova plataforma para implementação de *clusters*: o *cluster* RedBlue. Este utiliza uma árvore de sincronização de tarefas onde os nós são representados pelas cores vermelha e azul, o que motivou o nome da plataforma. Além disso, ela foi desenvolvida na linguagem Ruby, que possui acrônimo Rb, as mesmas iniciais de *red* e *blue*.

O RedBlue permite combinar o poder computacional dos laboratórios de informática existentes, ao mesmo tempo que dispensa conhecimentos em computação distribuída por parte dos usuários. Para ser utilizada, o usuário precisará saber apenas os métodos para leitura e escrita pelo terminal e manipulação de *strings*. A ferramenta possibilita que se desenvolvam aplicações utilizando diversas linguagens de programação, ampliando assim o público usuário.

Dessa forma, espera-se que a plataforma contribua para a aplicação da simulação computacional no ensino e na pesquisa, fomentando o desenvolvimento de habilidades necessárias ao engenheiro moderno. Ao mesmo tempo, espera-se prepará-los para utilizar todo o potencial computacional dos laboratórios de informática para o desenvolvimento de pesquisas científicas.

1.1 Objetivos

Nesta seção são apresentados os objetivos gerais e específicos deste trabalho.

1.1.1 Objetivo Geral

Este trabalho tem por objetivo apresentar uma plataforma que facilite a construção e utilização de *clusters* em laboratórios de informática para execução de simulações computacionais.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Ampliar o acesso a plataformas de alto desempenho a usuários com conhecimentos básicos em programação;
2. Reduzir a complexidade do desenvolvimento de *software* em ambientes paralelos e distribuídos;
3. Reduzir o tempo de execução das simulações computacionais;
4. Incentivar a adoção da simulação computacional como ferramenta de ensino e pesquisa;
5. Diminuir a ociosidade de laboratórios de informática.

1.2 Contribuições

Neste trabalho é apresentada uma plataforma para desenvolvimento e execução de aplicações distribuídas sobre *cluster* de computadores utilizando laboratórios de informática. Essa foi desenvolvida tendo em mente as necessidades e restrições da utilização de simulações computacionais como ferramenta de pesquisa e ensino em engenharias. Ela permite reduzir a complexidade do desenvolvimento de simulações computacionais, além de reduzir seu tempo de execução. Também não é necessário instalar programas nos computadores dos laboratórios para utilizá-los como *cluster*, uma vez que o sistema operacional e suas dependências são fornecidos pela rede. Dessa forma, é possível habilitar o modo *cluster* apenas quando necessário, deixando o laboratório disponível para outras atividades.

Para isso, diferentemente da abordagem tradicional, a plataforma faz uma separação clara entre as regras de negócio da aplicação cliente e rotinas administrativas de um *cluster*. Isso foi possível através do desenvolvimento de uma arquitetura em camadas com responsabilidades bem definidas. Assim é possível escrever aplicações em diferentes linguagens de programação, além de esconder detalhes de implementação do *cluster* para o usuário. A interação entre usuário, *cluster* e aplicação cliente é feita em alto nível por meio de uma interface em formato JSON e um pequeno conjunto de chaves reservadas utilizadas para controle do *cluster*. Esses dados são utilizados para construção da árvore de tarefas, uma estrutura que controla a ordem relativa de execução entre trechos paralelos e sequenciais e o fluxo de dados entre as tarefas. A partir dessa árvore se obtêm tarefas aptas a execução que são submetidas a uma lista de computadores disponíveis mantida pelo *cluster*. A separação desses mecanismos permite que o *cluster* se adapte a qualquer quantidade de computadores automaticamente.

Essas características foram pensadas para simplificar o desenvolvimento de simulações computacionais aplicadas a *clusters* de computadores. Ao longo deste trabalho

os mecanismos que possibilitaram que a plataforma incorporasse tais características serão descritos com mais detalhes. Dessa forma, além da utilização da plataforma em instituições de pesquisa e ensino, almeja-se que as ideias expostas neste trabalho possam ser utilizadas para melhoramento de outras iniciativas na área.

1.3 Organização do Texto

Este trabalho está organizado em 6 seções. A Seção 1 introduz o tema simulações computacionais e seu relacionamento com a indústria, pesquisa e ensino. São abordadas algumas barreiras quanto à sua utilização devido a necessidade de desempenho computacional e complexidade de desenvolvimento de aplicações. São apresentadas algumas soluções para esses problemas. A Seção 2 apresenta conceitos básicos sobre simulações computacionais, suas aplicações, vantagens e desvantagens. Além disso, apresenta conceitos e definições relacionadas a computação de alto desempenho utilizando *clusters*. São abordadas algumas características de mecanismos importantes de sistemas Linux, como gestão e intercomunicação de processos, necessários para a construção de um *cluster* de computadores. A Seção 3 apresenta a arquitetura da plataforma RedBlue e descreve os mecanismos de comunicação, estrutura de controle, distribuição de tarefas e gestão de computadores. Nessa seção também são listados os recursos de *software* e *hardware* utilizados para o desenvolvimento da plataforma. A Seção 4 descreve os experimentos utilizados para testar a plataforma, as condições em que os testes foram realizados e os resultados obtidos. A Seção 5 discute os resultados obtidos e sua relação com a arquitetura desenvolvida. Apresenta explicações adicionais sobre o desenvolvimento dos testes e cita vantagens e limitações da plataforma. A Seção 6 apresenta as considerações finais e as propostas de trabalhos futuros.

2 REVISÃO DA LITERATURA

Esta seção cobre os conceitos teóricos necessários para o entendimento deste trabalho. São abordados os seguintes assuntos: simulação computacional e suas aplicações; sistemas distribuídos; métricas de desempenho; escalonamento de processos; técnicas de intercomunicação entre processos; e serialização de dados.

2.1 Simulação Computacional

Simulação computacional é uma metodologia poderosa para projetar e analisar sistemas complexos. A simulação busca representar as características dinâmicas de um sistema real por meio de um modelo computacional (ROBINSON, 2004).

Na literatura encontram-se várias definições para simulação computacional. Por exemplo, "o uso do computador para imitar as operações de um processo do mundo real, considerando os relacionamentos lógicos, estatísticos ou matemáticos desenvolvidos em um modelo" (MCHANEY, 2009, tradução nossa). Outros autores levam em consideração a simplificação da realidade, a variabilidade do sistema e o propósito da simulação, definindo-a como: "experimentação em um computador da imitação simplificada de um sistema, bem como seu progresso através do tempo, com propósito de melhor entender e/ou melhorar o sistema" (ROBINSON, 2004, tradução nossa).

Ambas as definições utilizam os termos modelo e sistema, indicando que o modelo é uma imitação do sistema, que por sua vez é o objeto de estudo. Logo, faz-se necessário definir esses termos. Singh (2009) define sistema como uma coleção de entidades inter-relacionadas que atuam de forma colaborativa por algum propósito. Alguns exemplos de sistema são: uma célula, um átomo, uma galáxia, uma universidade, um avião, entre outros (VELTEN, 2009; SINGH, 2009). Singh (2009) define modelo como uma abstração do sistema (física, matemática ou computacional) que possui as propriedades e funções do sistema em que se baseia.

Os resultados de uma simulação podem ser manipulados através da variação de um conjunto de parâmetros de entrada. A variação controlada dos parâmetros de entrada um por vez, por exemplo, permite mapear o impacto do parâmetro no resultado obtido pela simulação (MCHANEY, 2009). Isso é útil para entender a dinâmica do sistema e identificar quais parâmetros mais influenciam a simulação. Em alguns casos, parâmetros com pouca influência sobre a simulação podem ser descartados a fim de simplificar o modelo computacional (VELTEN, 2009). Normalmente os resultados da simulação são interpretados por meio de algum tratamento matemático ou estatístico, como acontece com outros tipos de experimentos (MCHANEY, 2009). As informações obtidas após a interpretação são utilizadas para estimar as características do sistema, possibilitando refinar o modelo computacional.

O processo de desenvolvimento de uma simulação computacional pode ser dividido em algumas etapas (VELTEN, 2009):

1. Definições - Definição do problema a ser resolvido e as questões a serem respondidas. Definição de qual será o sistema em que o modelo se baseará.

2. Análise do Sistema - Identificação das partes relevantes do sistema para resolução do problema ou questão proposta. Normalmente essa fase consume muito tempo. Envolve pesquisa, revisão da literatura, consultorias entre outros meios de levantamento de dados sobre o sistema em estudo.
3. Modelagem - Desenvolvimento do modelo do sistema baseado nos resultados obtidos na fase de análise. Envolve a identificação do *software* adequado ou desenvolvimento de um próprio.
4. Simulação - Aplicação do modelo ao problema ou questão. Levantamento de estratégias que resolvam o problema ou respondam a questão.
5. Validação - Análise sobre as respostas obtidas. Comparação dos resultados com os dados obtidos na fase de análise. A informação é confiável? Se aplica ao sistema real (caso exista)? Quais são as limitações?

Essas etapas podem ser aplicadas ciclicamente quando se verifica falhas ou necessidades de aprimoramento no modelo proposto. Por exemplo, quando na fase de validação se conclui que o modelo não é adequado. Nesse caso todas as fases serão refeitas (VELTEN, 2009).

A dificuldade dos problemas tratados na ciência e engenharia é tipicamente originada da complexidade dos sistemas considerados. Nesse cenário, os modelos computacionais são uma ferramenta adequada para quebrar essa complexidade e tornar o problema tratável (VELTEN, 2009). O Quadro 1 elenca algumas situações em que a simulação computacional é adequada segundo McHaney (2009).

A seguir serão apresentados: um breve histórico sobre o surgimento da simulação computacional; algumas aplicações na indústria; aplicações na pesquisa; um apanhado sobre a utilização de simulações computacionais como ferramenta de aprendizagem; e as vantagens e desvantagens da simulação computacional.

2.1.1 Breve histórico

Os primeiros esforços para desenvolvimento de simulações remetem a Segunda Guerra Mundial. Nesse período, Jon von Neumann e Stanishlaw Ulam desenvolveram a técnica de simulação Monte Carlo, que ajudou na compreensão e no desenvolvimento da bomba atômica (MCHANEY, 2009). Na mesma época o exército do EUA buscava uma forma de acelerar os cálculos dos ângulos de disparo de sua artilharia. Essa tarefa foi assumida pela Escola Moore de Engenharia Elétrica na Universidade da Pensilvânia, acarretando a construção do primeiro computador eletrônico, o ENIAC (ROJAS; HASHAGEN, 2001), terminado em 1946. Essa primeira versão possuía uma forma de programação rudimentar, interconexão de unidades pré-programadas via cabos para resolução de um problema específico e configuração dos vários seletores de cada unidade, o que limitava sua flexibilidade (HAIGH; PRIESTLEY; ROPE, 2014a). No entanto, esse computador permanecia sendo o mais poderoso da época, razão pelo qual ele foi remodelado. A modificação foi finalizada em 1948, incorporando avanços na arquitetura de computadores do período (NEUMANN, 1945), tornando o ENIAC o primeiro computador

Quadro 1: Situações adequadas à simulação computacional

Situação	Exemplos
O sistema real ainda não existe e construir um protótipo é muito caro, consome muito tempo ou é perigoso.	Aeronave, Sistema de Produção, Reator nuclear.
É impossível construir o sistema.	Economia Nacional, Sistema Biológico
O sistema real existe, mas fazer experiências com ele é muito caro, perigoso ou prejudicial ao próprio sistema.	Proposta de mudanças em: Linha de montagem, Unidade Militar, Sistema de Transporte Público, Sistema de Esteiras (aeroporto).
É necessário fazer previsões para analisar um período longo de tempo em um pequeno tempo de estudo.	Crescimento populacional, Propagação do fogo em florestas, Estudos de Urbanização, Propagação de Epidemias
A modelagem matemática não tem uma solução analítica ou numérica prática.	Problemas Estocásticos, Equações Diferenciais Não-Lineares.

Fonte: McHaney (2009). Adaptado.

com suporte ao "paradigma moderno de programação". Esse termo descreve o mecanismo de controle dos computadores modernos, incluindo a execução automática de programas alocados em memória endereçável e instruções de máquina com argumentos (HAIGH; PRIESTLEY; ROPE, 2014c). Em abril e maio daquele mesmo ano aconteceram as primeiras simulações Monte Carlo computadorizadas da história. Simulou-se a cadeia de reações das partículas atômicas envolvidas durante uma explosão nuclear. Essa também foi a primeira vez que um programa escrito sobre o novo paradigma foi executado em um computador (HAIGH; PRIESTLEY; ROPE, 2014b).

Segundo McHaney (2009), outros acontecimentos importantes para evolução da simulação computacional foram:

- surgimento de novas linguagens de uso geral (assembly e FORTRAN) e algumas concebidas para criação de simulações, como GPSS (*General Purpose Simulation System*), SIMSCRIPT, SIMULA, GASP, SLAM e SIMAN, entre outras;
- documentação e publicação das técnicas utilizadas nos experimentos computacionais;
- estabelecimento de *workshops* e conferências para compartilhar as descobertas na área da simulação computacional (*Winter Simulation Conference*);
- advento dos computadores pessoais;
- expansão das linhas de *softwares* especializados para simulação, como pacotes de animação, ferramentas para desenvolvimento e melhoria das linguagens de simulação existentes.

- criação de *softwares* para simulação com interface gráfica;
- surgimento de modelos baseados em agentes, o que demonstrou que é possível derivar um comportamento global complexo por meio de interações locais simples.

Atualmente a simulação computacional é indispensável no estudo de áreas como: astrofísica, física das partículas, ciência dos materiais, engenharia, mecânica dos fluidos, estudo do clima, biologia evolucionária, ecologia, economia, teoria da decisão, medicina, entre outros (WINSBERG, 2015).

2.1.2 Simulação computacional na indústria

A simulação computacional é uma ferramenta extremamente importante para a indústria. Ela permite projetar e testar produtos, equipamentos e até processos do sistema produtivo antes que esses sejam implementados (NEGAHBAN; SMITH, 2014). Dessa forma é possível identificar muito dos problemas com antecedência, diminuindo o retrabalho e consequentemente reduzindo custos. Por essa e outras razões apresentadas anteriormente, a simulação computacional é utilizada em vários setores da indústria.

Na indústria automobilística a simulação computacional é usada para antever o arrasto aerodinâmico de veículos antes da fabricação de um protótipo (JANSSON; HOFFMAN; NAZAROV, 2011). O arrasto aerodinâmico tem impacto direto no *design* dos veículos, uma vez que possui forte influência sobre o consumo de combustível. A indústria automobilística também utiliza simulações computacionais para estudar os danos causados por colisões, os testes de impacto (*crash test*) (BOHN *et al.*, 2013). São realizadas simulações de impactos em diferentes ângulos e velocidades. Esses dados são utilizados para medir a deformação do veículo e as forças que agem sobre os passageiros durante o impacto, tornado possível a construção de carros mais seguros. Em ambos os casos, a simulação reduz o número de protótipos necessários para os testes, barateando o processo produtivo.

A indústria naval também faz intenso uso da simulação computacional, principalmente aplicadas a dinâmica dos fluidos (CFD - *Computational Fluid Dynamics*) (STERN *et al.*, 2013). Os navios são projetados utilizando modelagem baseada em simulação (SBD - *Simulation-Based Design*). Esse tipo de *software* permite avaliar o desempenho hidrodinâmico do modelo baseado em um conjunto de restrições pré-definidas. Assim é possível comparar quantitativamente duas opções de *design* durante a fase de projeto. Essa metodologia permitiu reduzir o número de protótipos necessários, sendo esses utilizados apenas nas fases finais do projeto. A simulação computacional permite estudar o efeito das ondas sobre o casco da embarcação em diferentes situações, tornando viável construir navios mais estáveis mesmo sobre condições de turbulência.

A indústria da construção civil utiliza simulações computacionais para estudar a interação entre pessoas e edificações (TANG; REN, 2008). Por exemplo, em ambientes com grande concentração de pessoas, como uma estação de metrô ou um estádio de futebol, a localização e quantidade de saídas é um item fundamental para evacuação em caso de catástrofes. Um tipo de simulação computacional chamada simulação baseada em agentes é utilizada nesses

casos para modelar o comportamento do ser humano. Os agentes, no caso pessoas virtuais, são capazes de tomar decisões, as vezes ruins ou aleatórias. Essas ações se assemelham as decisões tomadas por pessoas em situações de pânico, como as causadas pelo medo das chamas ou pela perda de visibilidade em um incêndio. Assim, os projetistas podem simular incidentes e medir os danos causados as pessoas e edificações sem o risco de causar ferimentos, o que poderia ocorrer em uma simulação com pessoas reais. Essas informações são utilizadas para melhorar o projeto, reduzindo constrições para acelerar a evacuação em casos de emergência.

A indústria de microprocessadores faz uso de simulações computacionais para obter a melhor configuração da localização dos componentes eletrônicos em pastilhas minúsculas (CHEN *et al.*, 2013). A arquitetura deve levar em consideração restrições de desempenho, consumo de energia, temperatura e confiabilidade, chamadas *Design Space Exploration (DSE)*. À medida que novos componentes são adicionados o número de ligações entre esses cresce, tornando o projeto mais complexo. Nesse contexto, a simulação computacional é utilizada para avaliar a qualidade do projeto para cada configuração desenvolvida.

A simulação computacional está presente em várias outras indústrias e nessas é aplicada em diferentes fases do processo produtivo (MOURTZIS; DOUKAS; BERNIDAKI, 2014). Os exemplos citados são apenas uma pequena amostra da utilização dessa ferramenta na indústria.

2.1.3 Simulação computacional na pesquisa

Atualmente as pesquisas científicas fazem uso extenso de computadores. Muito disso se deve à complexidade envolvida nas pesquisas. Os problemas tratados possuem uma enorme quantidade de fatores que influenciam o comportamento do sistema em estudo. Consequentemente, a interação entre esses fatores pode gerar um conjunto de relacionamentos muito grande, tornando-o tratável apenas computacionalmente. Nesse cenário, a modelagem e simulação computacional é utilizada como ferramenta para análise e simplificação de sistemas extremamente complexos. A seguir são apresentados três exemplos de utilização de simulações computacionais na pesquisa.

A meteorologia é uma das ciências que faz extenso uso de simulações computacionais. Os cientistas utilizam modelos atmosféricos para estudar fenômenos naturais, como a formação de furacões, e para prever as condições climáticas baseados no estados anteriores da atmosfera. Pelo menos três razões para a necessidade de se utilizar simulações computacionais para estudar fenômenos meteorológicos podem ser apontadas (PARKER, 2014): impossibilidade de repetição do evento; dificuldade para coleta de dados; e a grande quantidade de cálculos envolvidos na análise. Dessa forma, em muitos casos, é preferível realizar experimentos climáticos em um laboratório. Assim é possível controlar as variáveis que influenciam o estudo, repeti-lo várias vezes com diferente modelos e utilizar ferramentas de visualização sofisticadas para observar com mais detalhes o item de interesse.

O estudo das moléculas também utiliza simulações computacionais. Um sistema de moléculas pode ser formado por milhares de átomos que se movem em grande velocidade,

tornando difícil prever sua localização. Um tipo de simulação chamada Dinâmica Molecular é usada para investigar a estrutura das moléculas e as relações microscópicas entre essas (ALLEN, 2004). Essa técnica permite superar as restrições causadas pelo tamanho e a curta duração dos eventos em estudo. Também possibilita testar situações difíceis ou impossíveis de se obter, como experimentos reais realizados com pressão ou temperaturas extremas.

Atualmente as pesquisas sobre o desenvolvimento e possíveis tratamentos para o câncer também utilizam simulações computacionais (GARLAND, 2017). O câncer é uma doença caracterizada pela reprodução desordenadas das células, causada por mutações genéticas. Os cientistas procuram entender quais situações levam uma célula saudável e se transformar em um célula cancerígena. Para isso são utilizadas simulações computacionais para estudar como as células funcionam. O objetivo é identificar a sequência de eventos que induz o surgimento de um determinado tipo de câncer. Dessa forma, ao entender a dinâmica do desenvolvimento do câncer, seria possível tratá-lo antes mesmo que ele apareça. No entanto, esta é uma tarefa muito complexa, visto que os processos químicos celulares são inter-relacionados, tornando muito grande o número de fatores que podem levar ao câncer.

2.1.4 Simulação computacional na educação

As diversas etapas necessárias para desenvolvimento e aplicação de simulações computacionais se assemelham muito ao processo de aprendizagem por descoberta científica (ECKHARDT *et al.*, 2013). Nesse método de aprendizagem o aluno deve fazer previsões, manipular, observar, interpretar e refinar o experimento (simulação) para obter conclusões. Sobre o ponto de vista construtivista, ao interagir e tirar suas próprias conclusões sobre o evento analisado, o aprendiz absorve e retem por mais tempo os conceitos aprendidos (RUTTEN; JOOLINGEN; VEEN, 2012). No entanto, esse entendimento não é unânime, o que motiva vários autores a investigarem formas de potencializar o aprendizado utilizando simulações computacionais.

Jong e Joolingen (1998) discutem o papel da simulação computacional como ferramenta de aprendizagem por um viés construtivista. Nesse contexto, o aprendiz é visto como um agente ativo no processo de aquisição de conhecimento. Os autores elencam as principais dificuldades dos aprendizes no processo de descoberta via simulações quanto a: geração da hipóteses; modelagem dos experimentos; interpretação dos resultados; e gerenciamento do aprendizado. Conclui-se que, para ser efetiva, a simulação computacional deve ser aplicada em conjunto com outras formas de suporte a aprendizagem. Isso porque em alguns casos as etapas envolvidas na criação da simulação são fonte de dúvidas e insegurança. Logo, para que sejam aplicadas corretamente se faz necessário esclarecimentos adicionais. Alguns empecilhos dos alunos apontados pelos autores foram: dificuldade para criar uma hipótese por não saber como ela deve se parecer; dificuldade em tratar os dados com tendência a manipulá-los para que confirme a hipótese; criação de experimentos inefetivos para testar a hipótese; manipulação desordenada das variáveis do problema; pouca experiência na criação, análise e comparação de gráficos; falta de planejamento na condução do processo de pesquisa; entre outros. Para mitigar esses problemas

os autores sugerem estratégias como: fornecimento de informação adicional sobre o assunto em estudo antes da realização do experimento; dicas sobre como manipular as variáveis envolvidas no experimento; explicações adicionais na própria aplicação em que a simulação está sendo executada; aulas com exemplos reais sobre como criar uma hipótese; fornecimento de hipóteses pré-definidas sobre o assunto para que o aluno possa criar a sua própria; incentivo a criação de notas sobre os processos e resultados obtidos após cada experimento proposto; utilização de ferramentas para criação de gráficos para facilitar as atividades de predição; planejamento prévio dos experimentos; utilização de atividades que relacionem conceitos teóricos aos resultados observados na simulação; entre outras.

Liao e Chen (2007) sintetizam e analisam estudos sobre aprendizagem com simulações computacionais buscando comprovar se de fato o uso dessa metodologia produz melhores resultados frente ao ensino tradicional. A pesquisa foi motivada pela divergência de resultados obtidos em pesquisas anteriores. Alguns autores concluíram que o uso de simulações computacionais traz ganhos significativos ao desempenho dos estudantes enquanto outros reportaram não haver diferenças entre essa abordagem e o ensino tradicional. Os experimentos descritos nos trabalhos verificados foram repetidos com alunos de Taiwan. Um grupo de alunos foi submetido ao ensino tradicional e outro utilizou simulações computacionais como ferramenta de ensino. Aplicou-se os instrumentos de análise descritos nas pesquisas selecionadas. Devido a heterogeneidade desses trabalhos, calculou-se o tamanho do efeito (*Effect Size*) para reduzir os resultados a uma escala comum. O trabalho concluiu que o uso das simulações computacionais como ferramenta de ensino produz melhores resultados do que o ensino tradicional.

Dalgarno, Kennedy e Bennett (2014) apresentam um estudo sobre o impacto no aprendizado da exploração livre de parâmetros em simulações computacionais em comparação a observação de um conjunto de parâmetros pré-definidos. Os estudantes foram submetidos a duas simulações computacionais, uma sobre o tema aquecimento global e outra sobre a concentração de álcool no sangue. A simulação foi concebida para incentivar os usuário a prever, observar e explicar os resultados. Os resultados da simulação foram apresentados em forma de gráficos. Cada estudante realizou uma simulação utilizando exploração em um tema e observação em outro tema. Antes de cada simulação os estudantes realizaram um teste de conhecimento sobre o tema, que consistia em questões de múltipla escolha. Algumas questões tinham mais que uma resposta correta. Segundo os autores essas foram inseridas para se ter certeza que o conteúdo proposto foi entendido. O mesmo teste foi aplicado ao fim da simulação. As ações realizadas pelos estudantes durante o experimento foram registradas para análise posterior. Os resultados apontaram que a exploração dos parâmetros melhorou o rendimento dos estudantes apenas quando foi realizada de maneira sistemática, um parâmetro por vez. Essa estratégia se mostrou superior as demais. Não foram verificadas diferenças entre pré e pós-testes para alunos que realizaram uma exploração aleatória. O rendimento dos alunos utilizando a estratégia de observação e exploração aleatória foi semelhante. A implicação principal ressaltada pelos autores é que é necessário construir simulações que guiem os usuários na exploração dos

parâmetros de forma sistemática. Além disso, o *software* poderia alertar o usuário ao perceber um comportamento aleatório na manipulação dos parâmetros.

Dasgupta (2016) apresenta um estudo que busca compreender como criar um ambiente de aprendizagem tecnológico que permita desenvolver habilidades em engenharia em estudantes do ensino médio. Desenvolveu-se um modelo computacional de encanamento residencial com interface gráfica. O objetivo da simulação era construir a rede hidráulica da residência mantendo a pressão nas torneiras em uma valor pré-definido. Os parâmetros disponíveis para simulação eram diâmetro do cano, comprimento e curvatura. Os resultados observáveis eram custo total e pressão em cada seguimento. O custo máximo poderia chegar até a 3000 dólares. Antes da simulação propriamente dita, foram apresentadas aos alunos noções do que seria um encanamento bom e ruim, as restrições da simulação, e os conceitos sobre a influência do diâmetro e comprimento dos canos na pressão da água. A simulação foi utilizada como ferramenta de testes e foi intercalada com discussões em sala de aula, períodos de trabalho individual e em grupo. No final do trabalho os estudantes entregaram o projeto final do encanamento e obtiveram um retorno do tutor sobre o resultado alcançado. Os resultados da pesquisa demonstraram que a maior parte do estudantes consegue observar a relação entre dois parâmetros e apenas metade deles conseguiu expressar a relação envolvendo três parâmetros. O autor sugere que a atividade modelagem foi crucial para o entendimento do sistema em estudo, visto que os alunos tinham pouco conhecimento anterior sobre o assunto. O autor também destaca que os alunos conseguiram desenvolver noções de prioridade na escolha, relacionadas com os ganhos e perdas envolvidos em decisões de projeto.

Ao observar os resultados obtidos nos trabalhos citados, é possível concluir que a simulação computacional possui grande potencial para estimular a compreensão sobre diversos temas. Além disso, os trabalhos relatam o fortalecimento de habilidades necessárias a resolução de problemas inéditos, essenciais para o mercado de trabalho atual. No entanto, a simulação computacional deve ser aplicada em conjunto a atividades de suporte ao aprendiz a fim de maximizar seus benefícios.

2.1.5 Vantagens e desvantagens da simulação computacional

De forma geral, algumas vantagens das simulações computacionais frente a outras abordagens são (ROBINSON, 2004; MCHANEY, 2009):

- Menor custo - Experiências físicas ou experimentação em sistemas reais são normalmente caras. No primeiro caso, a realização do experimento pode depender de insumos, equipamentos, pessoal e local adequado para sua realização. Cada um desses itens possui um custo associado, que pode ser elevado. Além disso, normalmente o material utilizado será consumido, por exemplo, reagentes químicos. Experimentos em sistemas reais podem acarretar na paralisação das operações. Não há garantia que o experimento seja bem sucedido, o que pode piorar o desempenho do sistema. Por exemplo, parar parte de uma linha de produção para adequações experimentais pode reduzir o desempenho produtivo, gerando prejuízos. Em ambos os casos o uso de simulações computacionais poderia reduzir

os custos, seja por não utilizar insumos ou por permitir testar modificações sem paralisar as operações.

- Obtenção de resultados em menos tempo - Além do tempo gasto para configurar o experimento, talvez seja necessário observar o comportamento do sistema por muito tempo para obter resultados. Numa simulação computacional é possível modificar a escala do tempo para obter os resultados mais rapidamente. Nesse caso o tempo gasto dependerá da complexidade do experimento e do poder computacional do equipamento utilizado. Uma experimentação rápida também permite que novas configurações do experimento sejam testadas em um curto espaço de tempo.
- Controle das condições do experimento - Quando se analisa várias opções é importante controlar as condições dos experimentos para que se possa fazer comparações diretas. Isso é difícil de se obter em sistemas reais, visto que algumas variáveis não são controláveis. Por exemplo, a frequência de chegada de pacientes em um hospital ou as condições climáticas em um voo teste. Em alguns casos, o experimento não pode ser repetido ou previsto com a antecedência necessária. Por exemplo: desastres naturais, como um tsunami ou terremoto; ou campanhas militares. Isso pode ser superado ao utilizar simulações computacionais, uma vez que é possível repetir o experimento várias vezes utilizando as condições mais convenientes.
- Permite detectar inconsistências antes da construção do sistema real - Ao testar diferentes configurações durante a simulação é possível verificar situações em que o desempenho do sistema é ruim. Isso permite reavaliar a modelagem e identificar futuros gargalos no sistema real. A correção das inconsistências na fase de modelagem evita o retrabalho, uma vez que possibilita a correção de defeitos antes da construção do sistema real.
- Encoraja a criatividade - Muitas vezes ideias que trariam melhorias significativas nunca são implementadas devido ao medo de falhas. A simulação computacional permite que as ideias sejam testadas em um ambiente sem riscos, incentivando o compartilhamento de ideias criativas para resolução de problemas.
- Melhora o conhecimento sobre o sistema - A simulação envolve a criação de um modelo que descreva o sistema. A fase de análise implica em pensar sobre todos os aspectos do sistema e os inter-relacionamentos envolvidos. Isso por si só já melhora o entendimento do sistema, uma vez que envolve pesquisa, questionamento e troca de informações entre uma equipe, muitas vezes, interdisciplinar. A modelagem cria a oportunidade para pensar sobre aspectos que talvez ainda não tenham sido considerados.
- Melhora a visualização e comunicação - Simulações visuais permitem exibir a dinâmica do sistema em estudo. Isso é útil para demonstrar conceitos a pessoas que não possuem conhecimentos aprofundados na área de estudo, mas precisam conhecer o dinâmica geral do sistema. Por exemplo, para convencer um grupo de investidores que um projeto é viável ou para repassar a visão geral do sistema para uma equipe multidisciplinar.

No entanto, a simulação computacional também possui algumas limitações. A lista a seguir cita algumas desvantagens e dificuldades dessa abordagem segundo Robinson (2004), McHaney (2009) e Yang, Koziel e Leifsson (2013)

- Custo - Embora o custo da simulação computacional seja mais barato em relação a um experimento real, o investimento inicial não pode ser desconsiderado. É necessário investimento em *software* especializado, *hardware* robusto para executar a simulação e pessoal qualificado para análise e desenvolvimento do modelo computacional. Caso a equipe não tenha a expertise necessária pode ser necessário contratar treinamentos ou um consultor, encarecendo a utilização da metodologia.
- Demora para se obter resultados - O tempo gasto desde a definição do sistema em estudo até as primeiras respostas pode ser longo dependendo da complexidade do problema. Além disso, o tempo gasto na execução da simulação pode ser demasiadamente longo. Uma alternativa é reduzir a complexidade do modelo computacional a fim de ter um resposta viável, sacrificando a acurácia dos resultados. Outra alternativa é investir em *hardware* mais robusto e/ou reduzir o problema em partes menores para execução paralela.
- Produz resultados aproximados - A abstração necessária para criação do modelo normalmente inclui alguma taxa de erro. Por exemplo, caso a simulação utilize algum tipo de entrada aleatória, como para gerar eventos, um pequeno erro estará associado ao resultado obtido. Logo os resultados gerados são aproximações. Por essa razão é necessário interpretar os resultados a fim de verificar se são ou não confiáveis.
- A validação é uma tarefa difícil - Na etapa de validação deseja-se comprovar se resultados obtidos na simulação são condizentes com o sistema real. Quando o sistema real não existe não há como fazer comparações quantitativas. Nesse caso é necessário utilizar a intuição e a opinião de especialistas, trazendo subjetividade à interpretação dos resultados.
- Requer expertise - Aplicar a simulação computacional requer várias habilidades. Além da capacidade de escrever ou utilizar um *software*, são necessários conhecimentos em matemática, estatística, modelagem conceitual, validação. Habilidades em comunicação, trabalho em equipe e gerencia de projetos também são desejáveis.
- Extrapolação dos resultados obtidos - Algumas vezes os resultados obtidos nas simulações são considerados como verdade absoluta pelos usuários. No entanto, o processo de simulação e construído por humanos e por essa razão está sujeito a falhas. Caso a simulação não corresponda as expectativas é necessário investigar todo o processo minunciosamente antes de validar a simulação.

Na tentativa de diminuir a demora em se obter os resultados, em virtude do tempo de simulação, e do custo de *hardware* robusto, pode-se utilizar um *cluster* de computadores.

2.2 *Cluster* de computadores

A computação em *cluster* surgiu no início dos anos 90, quando notou-se que a paralelização de processos por meio da utilização conjunta de computadores poderia suprir a necessidade de processadores mais poderosos (BAKER; BUYYA, 1999). Para os fins deste

trabalho, um *cluster* de computadores é um sistema distribuído formado por um conjunto de computadores interligados com o objetivo de processar tarefas de modo colaborativo. Essa cooperação distingue um sistema distribuído de uma rede de computadores isolados (TANENBAUM, 2002).

Nesse contexto, os computadores que compõem o *cluster* são chamados de nós. Esses são normalmente unidades completas dotadas de processadores, memória, interfaces de I/O e sistema operacional, que são conectados por uma LAN (*Local Network Area*) (PINHEIRO, 2005). Segundo Buyya (1999), os principais componentes da arquitetura de um *cluster* são:

- computadores de alto desempenho ou estações de trabalho;
- sistemas operacionais;
- redes, *switches* de alto desempenho e interfaces de rede;
- protocolos de comunicação;
- *softwares* para abstração de imagem única, bibliotecas de programação, ferramentas e ambientes para programação paralela;
- e as próprias aplicações desenvolvidas para este ambiente distribuído.

As seguir serão apresentados alguns conceitos, classificações e métricas necessários para compreensão e análise de desempenho de *cluster* de computadores. Por fim serão apresentadas algumas vantagens e desvantagens da utilização de *clusters* de computadores.

2.2.1 Classificação, conceitos e métricas

Existem diversas classificações para *cluster*. Uma delas é quanto a sua finalidade e características, como confiabilidade, distribuição de carga e *performance*. Logo esses podem ser divididos em 3 tipos básicos. Os *Clusters* de Alta Disponibilidade são construídos para prover serviços de maneira ininterrupta através do uso de redundâncias explícitas. Caso um nó falhe, outro entrará em operação, mantendo o serviço no ar. *Clusters* de Balanceamento de Carga são caracterizados pela distribuição de tráfego ou requisições entre seus nós que possuem os mesmos programas em operação. E o terceiro tipo, *Clusters* de Processamento Paralelo, caracterizado pela alta disponibilidade e *performance* obtida através da divisão de tarefas em outras menores, processadas em paralelo nos nós, simulando um supercomputador (PITANGA, 2003). Dentro dessa última se destaca o *cluster* da classe *Beowulf*.

O termo *Beowulf* não é novo, e suas características foram se desenvolvendo durante anos por meio da escolha de ferramentas capazes de tornar o trabalho paralelo possível. Porém, o primeiro grupo a criar um supercomputador a partir de computadores comuns foi o CESDIS (*Center of Excellence in Space Data and Information Sciences*), supervisionado pela NASA e idealizado pelos pesquisadores Thomas Sterling e Donald J. Becker (QUINN, 2004).

Um *cluster* da classe *Beowulf* é caracterizado pelo uso de computadores e redes comuns (BROWN, 2007). Normalmente é composto por um nó especial de controle, chamado nó mestre ou cabeça e vários nós de processamentos, chamados de nós escravos. Outra característica é o uso de *software open source*, como um sistema operacional baseado em Linux e plataformas e linguagens de programação não proprietárias.

Para entender e mensurar o desempenho de um *cluster* é necessário conhecer alguns conceitos da computação de alto desempenho, como: latência e largura de banda; sincronização ou coordenação; *overhead*; granulosidade; e escalabilidade. A definição e uma breve explicação desses termos são listadas a seguir:

- Latência e largura de banda - Referem-se, respectivamente, ao tempo de transmissão de mensagens e a quantidade possível de dados transmitidos por um meio em um intervalo de tempo, normalmente *bits* ou outro múltiplo por segundo (TANENBAUM, 2002). O tempo de transmissão é o tempo que a informação leva ao sair do emissor até ser recebida pelo receptor. Caso as aplicações que atuam como emissor e/ou receptor não sejam paralelizadas, provavelmente ocorrerá um bloqueio. Isso significa que a aplicação ficará parada até que a transmissão seja concluída. Uma vez que a rede normalmente é bem mais lenta, se comparada aos barramentos de um computador, a latência rede é um fator que reduz o desempenho de aplicações distribuídas.
- Sincronização - Refere-se a um ponto de coordenação necessário a fim de permitir a cooperação entre tarefas. Esse tipo de situação ocorre quando uma operação só pode iniciar após recuperar o resultado de outra operação que ainda não terminou de executar (TANENBAUM; STEEN, 2007). Situações desse tipo podem limitar o desempenho por restringir a quantidade de operações que podem executar em paralelo. No entanto, em certos casos, não há outra opção senão esperar a conclusão da operação anterior para evitar trabalhar sobre dados inconsistentes.
- *Overhead* - É o tempo gasto em operações administrativas necessárias para execução de uma tarefa (FISCHBORN, 2006). Por exemplo, para iniciar uma aplicação é necessário criar um processo. Ao medir o tempo de execução de uma aplicação a partir do momento em que o usuário a executa até seu término, o tempo de configuração será considerado. Nesse caso, o tempo de configuração foi um tempo desperdiçado (*overhead*), mas necessário para executar a aplicação. A paralelização de processos, troca de contexto de processos e transmissão em rede são outras operações que podem resultar em *overhead* elevado.
- Granulosidade - É a razão entre o tempo de operação útil em comparação ao *overhead* associado a sincronização e troca de mensagens em aplicações paralelas (RAUBER; RÜNGER, 2010). Por exemplo, caso o tempo de execução das partes que executam em paralelo sejam elevadas em consideração ao tempo gasto pela sincronização, será vantajoso executá-la no *cluster*. Nesse caso, a granulosidade da aplicação será considerada grossa ou alta. Porém, caso o tempo de execução da aplicação seja curto, alguns segundos, não será vantajoso aplicá-la ao *cluster*. Nesse caso o tempo gasto pela configuração da aplicação e troca de mensagens será mais alto que o tempo em que de fato essa estará em execução. Para esse cenário a aplicação será considerada de granulosidade fina ou baixa.
- Escalabilidade - Indica quanto um sistema pode ser expandido sem deterioração de desempenho (TANENBAUM; STEEN, 2007). Quando um *cluster* possui boa

escalabilidade, é possível adicionar vários computadores com baixa perda de desempenho. Porém, caso existam problemas de escalabilidade, ao adicionar mais máquinas, o desempenho diminuirá bruscamente.

2.2.2 Vantagens e Desvantagens de clusters de computadores

Como qualquer tecnologia, a adoção de um *cluster* possui vantagens e desvantagens. O Quadro 2 agrupa os pontos fortes e fracos da computação em *cluster* segundo os autores Rose e Navaux (2004), Bacellar (2010) e Real e Filho (2012).

Quadro 2: Vantagens e Desvantagens de um *cluster* de computadores

Vantagens	Desvantagens
Custo-benefício. Um cluster pode entregar desempenho similar a um supercomputador com menor custo.	Os programas devem ser escritos explicitamente para um ambiente distribuído, tornando-os mais complexos.
Maior facilidade de implantação e manutenção. Os componentes são facilmente encontrados no mercado. A produção em larga escala barateia o custo unitário.	É bom apenas para problemas paralelizáveis, uma vez que a melhoria no desempenho advém da distribuição e processamento paralelo de tarefas.
Expansibilidade. O desempenho pode ser melhorado bastando-se adicionar mais nós de processamento.	Aplicações de baixa granulosidade não terão *speedup* significativo devido ao tempo gasto na transmissão e gerenciamento de tarefas.
Maior tolerância a falhas. O sistema pode continuar funcionando mesmo que um nó falhe.	Aplicações com grande produção de dados são afetadas pela velocidade da rede, significativamente menor que a de um barramento.

Fonte: Rose e Navaux (2004), Bacellar (2010), Real e Filho (2012). Adaptado.

As principais vantagens da utilização de *clusters* são seu baixo custo, visto que podem ser construídos com computadores comuns, e a possibilidade de melhorar seu desempenho, adicionando mais máquinas. As principais desvantagens são o aumento da complexidade do código devido as características do ambiente distribuído em vários computadores e a necessidade de explicitamente paralelizar a execução de tarefas.

Uma vez que aplicações distribuídas são mais complexas do que as que executam em um único computador, é necessário ter uma melhor compreensão de onde a aplicação irá executar. Isso significa compreender como algumas partes do Linux funcionam, visto que variantes Linux são as mais utilizadas para essa finalidade (TOP500 Team, 2017). Essa compreensão será útil para aproveitar melhor o potencial dos computadores e para a escolha da tecnologia apropriada para o desenvolvimento do *cluster*.

2.3 O escalonador de processos do Linux

Quando um arquivo executável é iniciado ele cria um processo. Cabe ao escalonador do sistema decidir por quanto tempo o processador dará atenção a ele (*Time Sharing*). Normalmente a política *Round Robin* é utilizada, definindo uma quantidade de tempo que o processo estará alocado em um processador (HENNESSY; PATTERSON, 2007). Quando essa fatia de tempo é esgotada, o processador passa a dar atenção a outro processo. Essa operação é chamada de mudança de contexto. Outros eventos também provocam uma mudança de contexto, como uma interrupção para escrita em disco ou acesso a rede. Quando isso acontece, o processo é retirado do processador enquanto essas operações são realizadas. Esses mecanismos permitem que um processador com um único núcleo execute diversas tarefas, aparentemente simultâneas para o usuário.

Hoje, no entanto, é muito comum que os processadores tenham mais de um núcleo. Isso faz com que o escalonador tenha a tarefa adicional de escolher em qual núcleo o processo será executado. Por padrão, o escalonador do Linux tenta agrupar processos menos onerosos em um único núcleo, para economizar recursos, enquanto tenta dedicar núcleos a processos mais pesados (LOVE, 2013). Essa alocação é dinâmica, o que faz com que um processo possa trocar de núcleo durante a execução. Também por padrão, os processos de um usuário são criados com o mesmo nível de prioridade. Assim, pode ser que o escalonador não priorize alguma tarefa julgada importante para o usuário.

Caso a troca de contexto aconteça com frequência, o desempenho será ruim, pois constantemente ocorrerão *cache missings*, resultando em buscas na hierarquia de memórias (EIJKHOUT; CHOW; GEIJIN, 2016). Esta condição é chamada *Cache Thrashing*. Embora seja possível definir a prioridade de um processo frente a outros, a intervenção manual do usuário pode gerar panes no sistema. Isto porque um processo pesado com alta prioridade pode fazer com que outros processos essenciais ao sistema operacional não consigam ser alocados no processador.

Para que uma aplicação seja paralelizável, ela deve ser programada para isso. O sistema operacional não divide uma aplicação automaticamente, apenas escolhe o processador e aloca o processo. Logo, o programador deve saber como paralelizar suas aplicações.

2.4 Processamento Paralelo em Processadores Multinúcleos

Existem basicamente duas formas de executar tarefas em paralelo em um processador multinúcleos: via *threads* ou processos (RAUBER; RÜNGER, 2010). É necessário aqui distinguir a diferença entre concorrência e paralelismo. Quando duas atividades disputam o mesmo recurso e apenas uma pode utilizá-lo a cada momento temos um comportamento concorrente. Ao passo que quando duas atividades estão ativas simultaneamente utilizando recursos dedicados ou compartilhados temos um comportamento paralelo.

O sistema operacional é responsável por implementar o funcionamento das *threads*, o que pode variar de um sistema para o outro. Desta forma, a linguagem de programação utilizada deve ser capaz de se comunicar com a API (*Application Programming Interface*)

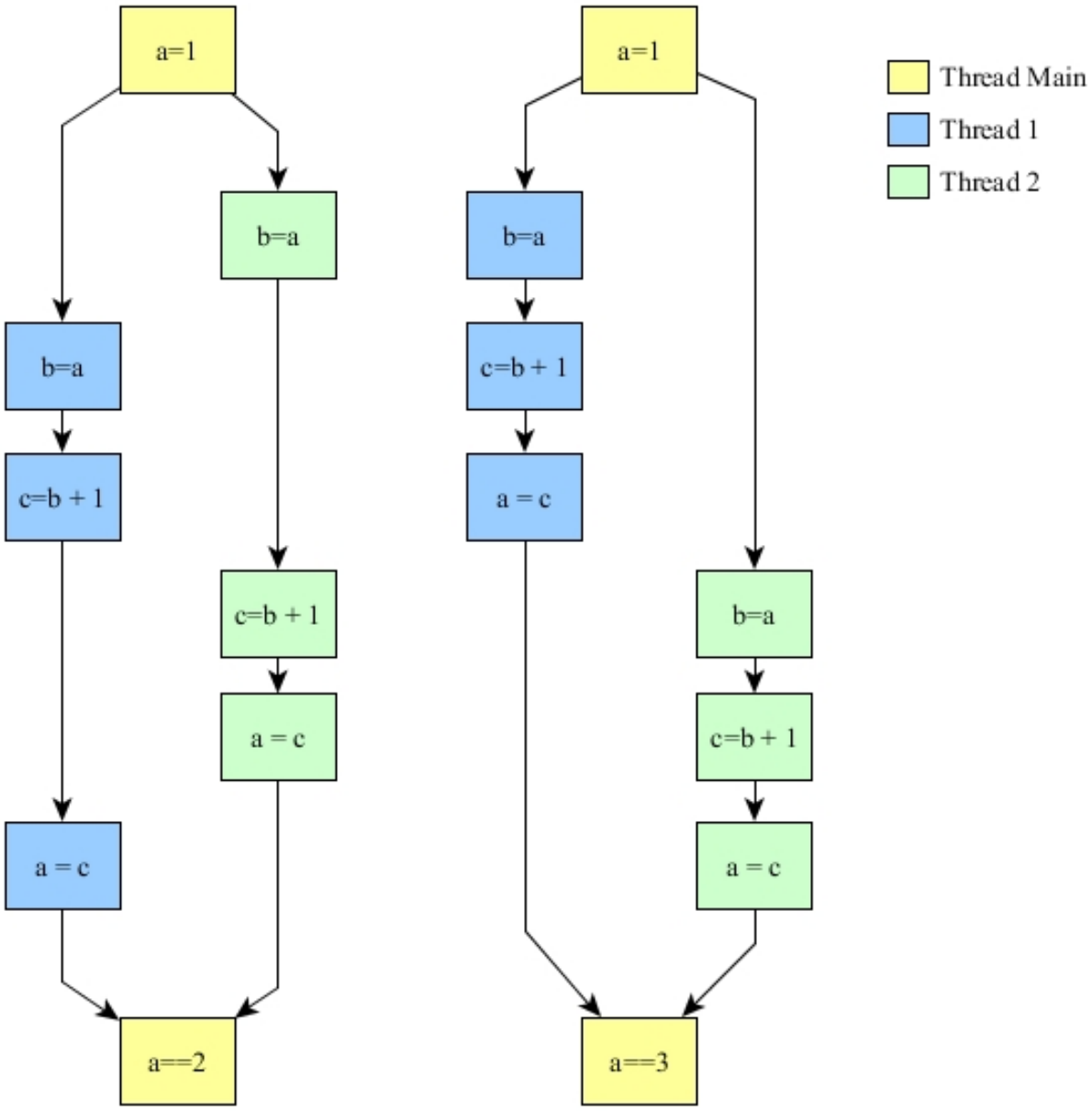
do sistema operacional para conseguir um desempenho nativo. Uma *thread* pode ser definida como um processo leve, capaz de compartilhar os recursos de um processo principal (*Thread Main*) com outras *threads* do mesmo grupo. As variáveis visíveis no escopo de uma parte de código paralelizado são compartilhadas entre as demais *threads*. Logo, o acesso as variáveis compartilhadas normalmente é paralelo para leitura e concorrente para escrita, enquanto os trechos de código são executados em paralelo. Esta abordagem permite utilizar os recursos de vários núcleos de processamento e ao mesmo tempo economiza memória, uma vez que este espaço é compartilhado. A principal desvantagem desse método é a complexidade adicionada ao código para prevenir que o acesso concorrente corrompa as variáveis compartilhadas (LOVE, 2013).

A fim de minimizar os problemas causados pelo acesso concorrente via *threads*, a implementação oficial da linguagem Ruby utiliza um semáforo global chamado GIL (*Global Interpreter Lock*) (COOPER, 2016). Esse mecanismo faz com que apenas uma *thread* seja executada por vez dentro de um processo. Dessa forma, em Ruby as *threads* têm comportamento concorrente, mas não são executadas em paralelo. Apesar disso diminuir as chances de um comportamento incorreto, não garante a atomicidade de trechos de códigos executados em *threads*. Um exemplo desta situação é exibido na Figura 1. A variável "a" é compartilhada em ambas as *threads*. Estas executam uma operação de soma utilizando variáveis intermediárias para armazenamento e posterior atualização do valor de "a". Em "Escalonamento 1" é gerado um valor diferente do esperado (2) devido a separação das operações que utilizaram o valor desatualizado de "a". Já em "Escalonamento 2" as operações não foram divididas, gerando o resultado correto (3). Como o escalonamento das *threads* é decidido pelo sistema operacional, o código deve garantir a integridade dos valores compartilhados por meio de mecanismos sincronização. A não adoção desses mecanismos acarretará em *bugs* de difícil localização, uma vez que o comportamento incorreto será gerado de forma intermitente e a priori não serão geradas exceções.

Portanto, a única forma de se obter paralelismo real com a implementação padrão da linguagem Ruby é por meio da criação de múltiplos processos. A principal diferença da implementação de paralelismo via múltiplos processos é que não há compartilhamento de variáveis. Logo, cada processo possui seu próprio espaço de endereçamento em memória. Desta forma, para que haja colaboração entre processos é necessário utilizar troca de mensagens enquanto que via *threads* isto é feito através do compartilhamento de memória.

Uma das formas de construir um novo processo é por meio da duplicação de processos. No Linux isto é possível via a operação *fork* (MATTHEW; STONES, 2009). Essa operação faz uma cópia exata do processo pai o qual recebe identificador do processo filho (PID). Utilizando-se a linguagem C, é possível identificar se um trecho de código esta executando no processo filho ou no processo pai verificando-se o valor de retorno da operação *fork*. Quando o valor de retorno é igual a zero, significa que se trata do processo filho. Caso seja maior que zero

Figura 1: Escalonamento de *threads*. Exemplo de uma situação que a ordem de execução das *threads* somada ao compartilhamento de variáveis modifica o resultado dos cálculos.



Fonte: Próprio autor.

indica que o trecho está sendo executado no processo pai. Isto permite que processo pai e filho executem operações diferentes.

Ao se comparar o custo computacional da criação de *threads* e processos, verifica-se que o segundo é mais dispendioso por envolver operações de cópia de memória. Isto foi minimizado com a implementação de *fork* utilizando-se a técnica *Copy-on-Write* (COW). Essa implementação faz com que o processo filho use um espaço de memória virtual, compartilhando o espaço de memória do processo pai. Este espaço compartilhado é mantido até que aconteça uma operação de escrita em memória, seja pelo processo pai ou pelo filho. Quando isto acontece, o verdadeiro espaço em memória é criado para o processo filho a fim de prevenir que as mudanças sejam visíveis em ambos processos (LOVE, 2013). A Figura 2 ilustra essa técnica.

O processo pai cria três variáveis de instância representadas como páginas em memória. Logo em seguida são criados dois novos processos utilizando *fork*. Neste momento, um espaço de memória virtual é criado nos processos filhos, apontando para posições reais de memória do processo pai. Nas operações seguintes, os processos filhos atualizam o valor de algumas variáveis que deixam de ser compartilhadas. No entanto, páginas inalteradas continuam sendo compartilhadas entre os processos. Com a aplicação desta técnica, o custo de criação de um novo processo por duplicação se torna mais próximo ao da criação de uma *thread*. Isto porque a criação do espaço privado de memória não precisa ocorrer necessariamente no momento da criação do novo processo. Além disso, torna possível que quantidade de memória utilizada seja menor devido a possibilidade de compartilhamento de páginas apenas para leituras.

Ao utilizar múltiplos processos para obter paralelismo se faz necessário a adoção de técnicas para troca de informação entre esses. Algumas dessas técnicas são descritas a seguir.

2.5 Técnicas de intercomunicação entre processos (IPC)

Para que seja possível a colaboração entre processos na realização atividades em paralelo são necessários mecanismos para troca de informação. Desta forma, a cooperação entre processos dependerá de técnicas de intercomunicação. No Linux, isto é possível com a utilização de algumas técnicas como: *Pipes*, *Named Pipes* (FIFOs), *Sockets* e *Signals*.

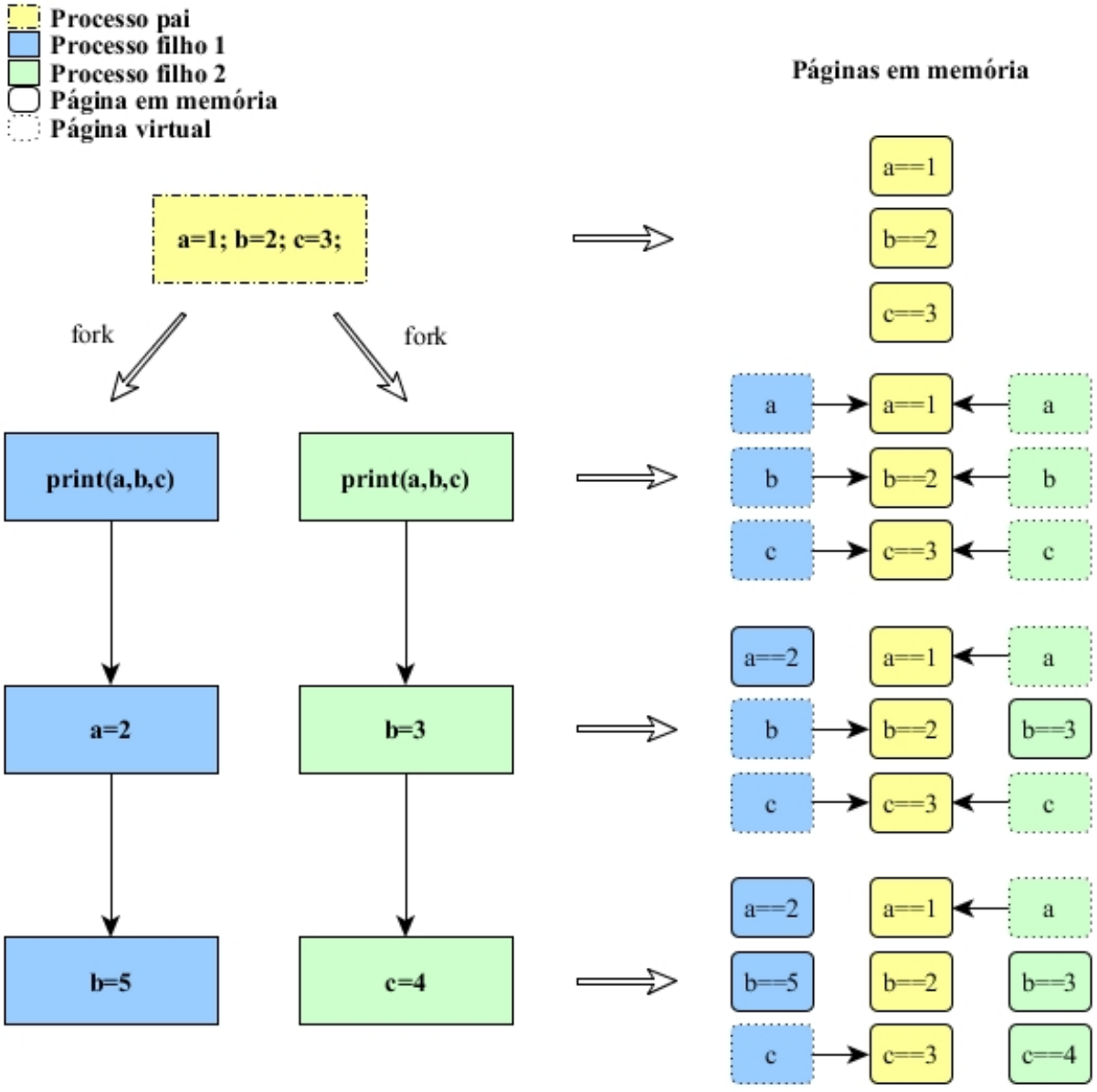
As subseções seguintes trazem um breve detalhamento sobre algumas das técnicas utilizadas para intercomunicação de processos no Linux.

2.5.1 Pipes

Pipes são a forma mais antiga de intercomunicação entre processos (KERRISK, 2010). Essa é baseada no compartilhamento de descritores de arquivos. Isto é alcançado através da duplicação do processo por meio da operação *fork*, onde processo pai e filho passam a utilizar os mesmos descritores. Um descritor de arquivo é um inteiro não negativo utilizado pelo *Kernel* para identificar um arquivo que foi aberto ou criado por um processo. Não existe o conceito de mensagem ou pacote quando se utiliza *pipes*, a transmissão é baseada em um fluxo de dados de tamanho indeterminado que serão lidos na mesma ordem em que foram inseridos. Ou seja, o acesso é sequencial.

Figura 2: Aplicação da técnica *Copy On Write*

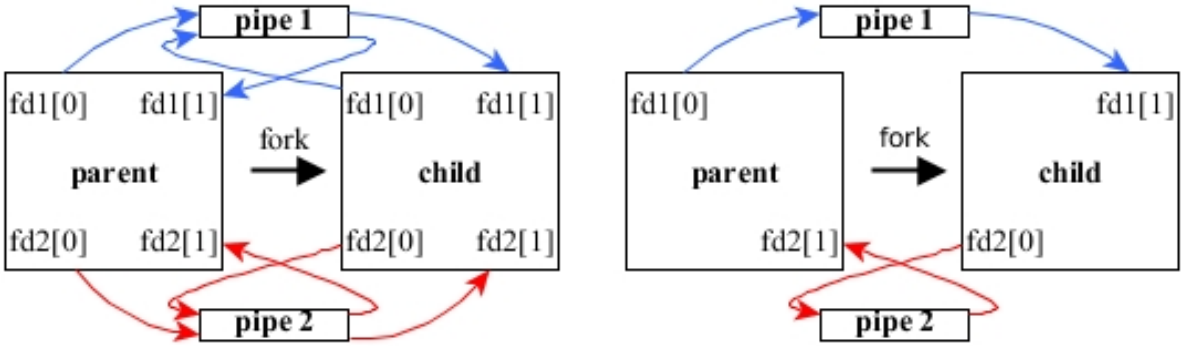
.



Fonte: Próprio autor.

Os *pipes* possuem duas limitações: são *half-duplex* e podem ser usados apenas entre processos com um ancestral em comum, normalmente entre pai e filho. Neste contexto, *half-duplex* significa que um descritor de arquivo pode ser utilizado somente para leitura ou somente escrita em cada processo. Logo, para implementar a comunicação em ambas as direções são necessários dois *pipes*. A Figura 3 ilustra o processo de comunicação entre dois processos utilizando *pipes*. No primeiro momento os canais de comunicação estão duplicados. Desta forma é necessário fechar alguns descritores para determinar a direção do fluxo de dados em cada *pipe*. Neste exemplo, o "pipe 1" é o canal utilizado para que o processo pai se comunique com o filho e o "pipe 2" é responsável pela direção contrária.

Figura 3: Comunicação entre processos utilizando *Pipes*.



(a) Momento 1

(b) Momento 2

Fonte: Stevens (1999). Adaptado.

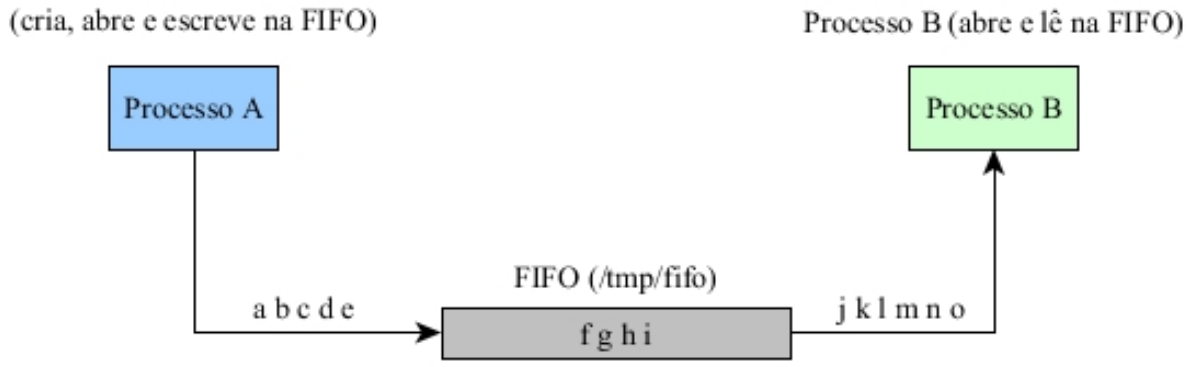
Normalmente a leitura de um *pipe* bloqueia a execução do processo até que este receba uma instrução de fim de arquivo ou até que existam dados para serem lidos. Isto implica que caso nenhum dado seja escrito na outra extremidade do *pipe*, o processo ficará parado indefinidamente (*Dead Lock*). Isto também pode acontecer em operações de escrita. Neste caso, os dados são escritos no *buffer* do emissor até que este bloqueie esperando que haja espaço disponível para novas escritas. Caso o processo receptor não consuma os dados, o processo emissor ficará parado indefinidamente. Essa situação pode ocorrer facilmente quando se trabalha com um fluxo grande de dados, uma vez que por padrão o *buffer* de um *pipe* possui tamanho máximo de 65536 *bytes*. Para que isto não ocorra é necessário implementar uma rotina para tratar possíveis exceções e sincronizar a produção e consumo de dados.

2.5.2 Named Pipes (FIFOs)

O funcionamento de um *Named Pipe* (FIFO - *First In, First Out*) é muito parecido com o de um *pipe*. A comunicação é feita através de fluxos de dados sequenciais, possuem um *buffer* de dados de tamanho máximo fixo, e permitem operações com e sem bloqueio. Sua principal diferença é que este pode trocar informações entre processos não relacionados (STEVENS; RAGO, 2013). Isto é feito através da utilização de um tipo especial de arquivo

chamado FIFO (*First In First Out*). Este é criado utilizando-se o comando *mkfifo* passando-se como parâmetro o nome do caminho para a criação do arquivo e a máscara de permissões de acesso. Logo, o caminho para o arquivo deve ser conhecido por ambos processos, que também devem possuir permissão para ler e/ou escrever na FIFO. Uma FIFO é similar a um arquivo regular no Linux, assim, as operações de I/O (*read, write, open, close*) funcionarão da mesma forma. As mesmas precauções para evitar *Dead Locks* em *pipes* também se aplicam aqui. Normalmente uma FIFO é compartilhada entre múltiplos processos para escrita. Neste caso deve se ter atenção especial sobre a atomicidade dessas operações. Para escritas que não ultrapassem a tamanho do *buffer* a atomicidade da escrita é garantida. Caso contrário o fluxo será interpolado com parte dos dados escritos por outros processos. A Figura 4 ilustra o funcionamento de uma FIFO.

Figura 4: Comunicação entre processos utilizando FIFO.



Fonte: Próprio autor.

2.5.3 Sockets

Os *sockets* são mecanismos que permitem a comunicação entre processos que estão executando em uma mesma máquina ou entre máquinas diferentes conectadas por uma rede. A API de *sockets* foi formalmente especificada no padrão POSIX.1g, baseado na interface de *socket* do sistema 4.4BSD (*Berkeley Software Distribuition*) (STEVENS; RAGO, 2013), a qual foi suplantada pela especificação SUSv3 (*Single UNIX Specification*) (KERRISK, 2010). Tipicamente um *socket* é usado para o cenário cliente-servidor, onde uma aplicação servidor cria um *socket* e o relaciona a um endereço para que a aplicação cliente possa localizá-lo.

Um *socket* é criado usando a função *socket()* com os argumentos domínio, tipo e protocolo, retornando em caso de sucesso um descritor de arquivo do tipo *socket*. Ou seja, um *socket* é visto pelo sistema operacional como um tipo especial de arquivo, desta forma algumas funções que atuam sobre arquivos regulares, como *read* e *write* também funcionarão sobre um descritor de *socket*. Este descritor não pode ser compartilhado, logo, um descritor de *socket* pertence a um único processo servidor.

O argumento domínio determina o formato de endereçamento, se a comunicação se dará entre aplicações na mesma máquina ou não, e por qual meio se dará a comunicação.

Os sistemas operacionais modernos suportam pelo menos os domínios UNIX (AF_UNIX) e de Internet, IPv4 (AF_INET) e IPv6 (AF_INET6) (KERRISK, 2010). O prefixo "AF_" significa *address family*, denotando a diferença de endereçamento entre esses domínios. No domínio UNIX a comunicação é feita através do *kernel*, portanto na mesma máquina, e o endereçamento se dá pela localização de um arquivo (*pathname*) na estrutura de diretórios. Para fins de portabilidade entre sistemas, se utiliza comumente criar este arquivo em "/tmp" ou "/usr/tmp" (MATTHEW; STONES, 2009). Para os domínios IPv4 (*Internet Protocol Version 4*) e IPv6 (*Internet Protocol Version 6*) a comunicação acontece via rede utilizando respectivamente estes protocolos de rede. O endereçamento utiliza o endereço IP, nas respectivas versões, somado ao número da porta utilizada, o que permite ao Linux localizar e rotear as conexões para o processo correto.

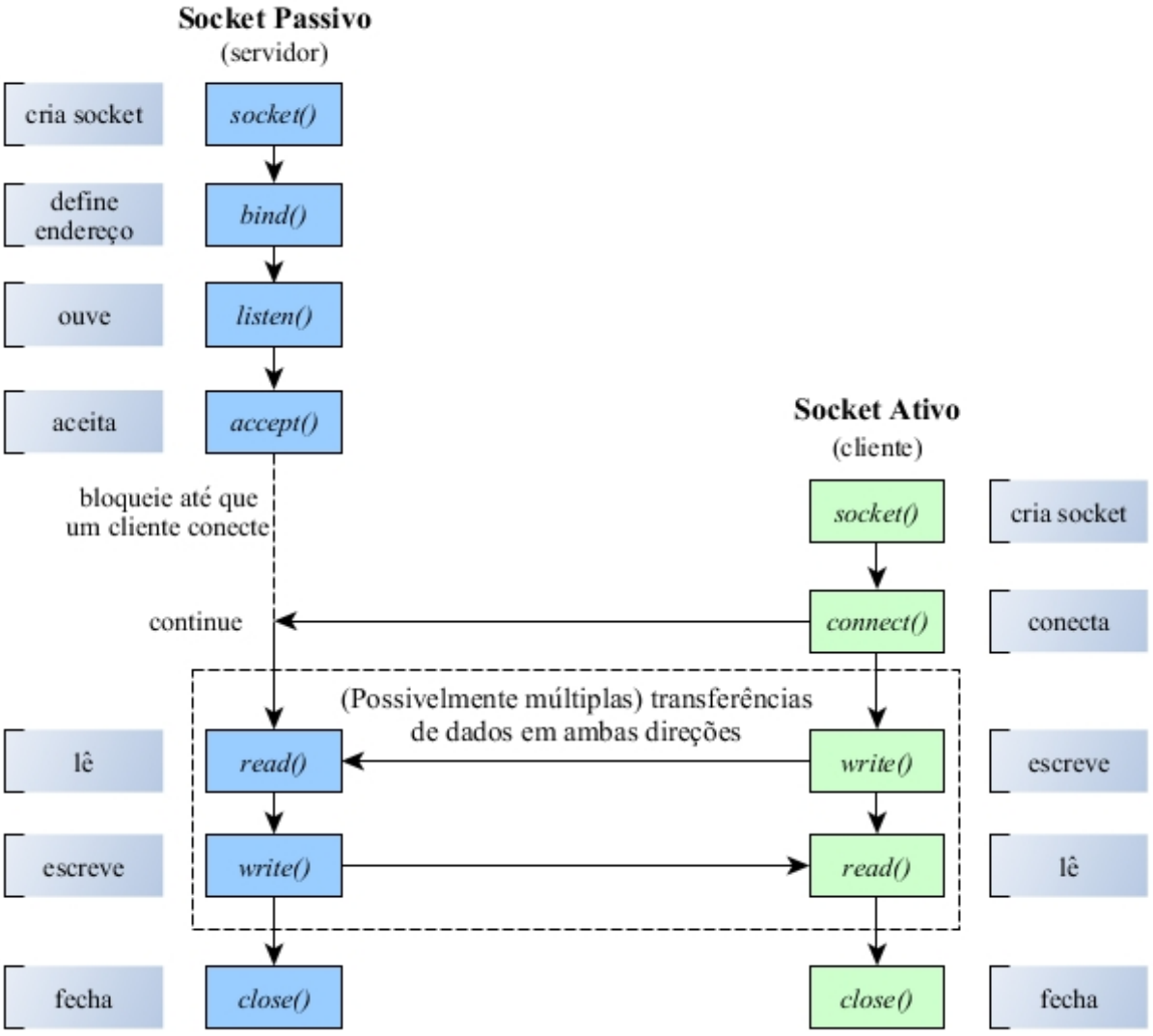
O argumento tipo determina as características da comunicação as quais definem o tipo de *socket*. Qualquer implementação prove pelo menos os tipos *stream* (SOCK_STREAM) e *datagram* (SOCK_DGRAM), os quais são suportados pelos domínios citados anteriormente (KERRISK, 2010). O tipo *stream* prove entregas garantidas, é bidirecional e sua comunicação é baseada em fluxo de bytes assim como *pipes*. Este também é orientado à conexão e precisa estabelecer um vínculo entre os pares antes de enviar uma carga útil de dados. O tipo *datagram* faz a comunicação utilizando o conceito de mensagens de tamanho fixo chamadas datagramas. A entrega das mensagens não é garantida, fazendo com que seja possível que as mensagens cheguem fora de ordem, duplicadas ou que não sejam recebidas. No entanto, não é necessário estabelecer uma conexão para enviar os dados, já que cada mensagem leva consigo o endereço de entrega.

O argumento protocolo permite escolher um protocolo alternativo, caso o domínio e o tipo especifiquem mais de uma possibilidade. Ao utilizar o valor 0 (zero) neste argumento, seleciona-se o protocolo padrão. O protocolo padrão para o domínio AF_INET com tipo SOCK_STREAM é o TCP (*Transmission Control Protocol*). Este protocolo é capaz de fragmentar e reordenar mensagens longas, além de possuir mecanismos de retransmissão em caso de perda de pacotes. Para o domínio AF_INET com tipo SOCK_DGRAM utiliza-se como padrão o protocolo UDP (*User Datagram Protocol*), sem garantia de entrega e reordenação de pacotes (MATTHEW; STONES, 2009). O domínio AF_UNIX não utiliza um protocolo de rede para transmitir os dados como nos outros domínios. Os dados são transmitidos diretamente através do *kernel* sem a necessidade de gerenciamentos adicionais, como análise de cabeçalho ou número de sequencia, tornando-o o tipo de *socket* mais eficiente para comunicações locais. Como nos demais domínios de *sockets*, são providas as interfaces para *streams* e datagramas, mas ambos com entrega confiável (STEVENS; RAGO, 2013).

A Figura 5 exibe as chamadas de sistema em *sockets* baseado em fluxo de dados. A função *socket()* é usada para criá-lo. Então a função *bind()* relaciona um endereço para o *socket* que passa a monitorá-lo utilizando a chamada *listen()*, mantendo-se bloqueado. O processo servidor fica então aguardando alguma conexão, utilizando a chamada *accept()* para aceitá-la e de fato realizar a comunicação, fechando-a em seguida com a chamada *close()*. Do ponto de vista

do cliente, basta criar um *socket* e especificar o endereço do par para iniciar a comunicação com *connect()*. Nota-se que o funcionamento deste tipo de *socket* é semelhante ao de uma chamada telefônica, com etapas de identificação, estabelecimento de comunicação e transmissão de voz.

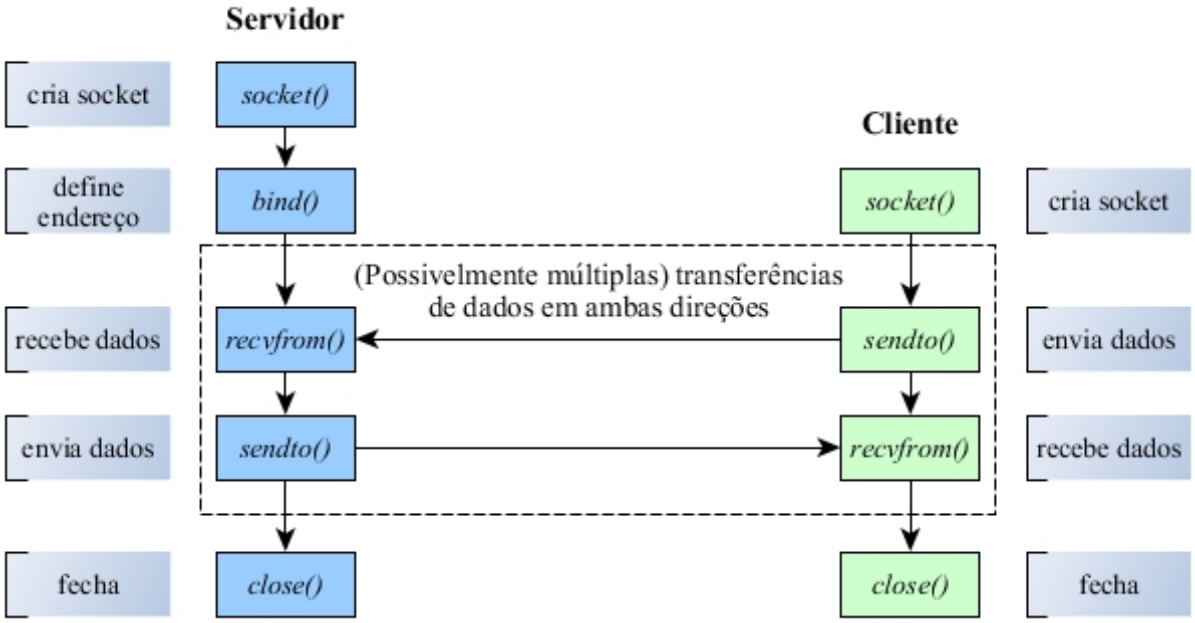
Figura 5: Funcionamento básico de *stream sockets*.



Fonte: Kerrisk (2010). Adaptado.

A Figura 6 exibe as chamadas de sistema para *sockets* baseados em mensagens. O processo cliente e servidor utiliza a chamada *socket()* para criá-lo. O servidor relaciona-o a um endereço com a chamada *bind()* e em seguida utiliza *recvfrom()*, bloqueando a execução até que uma mensagem chegue. O cliente utiliza a chamada *sendto()* para enviar uma mensagem, incluindo nesta o endereço do destinatário e remetente. Desta forma o servidor saberá para quem retornar a mensagem. Ao fim da troca de mensagens cliente e servidor utilizam a chamada *close()* para encerrar o *socket*. Neste caso o funcionamento se assemelha ao sistema postal, onde as mensagens são enderçadas independentemente.

Figura 6: Funcionamento básico de *datagram sockets*.



Fonte: Kerrisk (2010). Adaptado.

2.5.4 Signals

Um *Signal*, sinal, é uma interrupção de software que provê um meio de tratar eventos de forma assíncrona (STEVENS; RAGO, 2013). Esta técnica permite notificar um processo de uma condição ocorrida. Por exemplo, ao se usar a combinação de teclas de interrupção, normalmente "Ctrl + c", para parar um processo que está executando no terminal. Esta combinação dispara o sinal SIGINT no qual sua ação padrão é interromper o processo.

No entanto, ao receber um sinal o processo pode escolher entre três ações:

1. Ignorar o sinal. O que não é recomendado, pois exceções de hardware também utilizam sinais. Os sinais SIGKILL e SIGSTOP não podem ser ignorados.
2. Executar a ação padrão do sinal. Neste caso existem 5 ações padrão:
   a) O sinal será ignorado fazendo com que o Kernel o descarte.
   b) O processo será terminado de maneira anormal.
   c) Um arquivo *core dump* será gerado e então o processo será terminado. Este arquivo normalmente é utilizado por *debuggers*.
   d) O processo será suspenso e seu estado mudará para *stopped*.
   e) O processo anteriormente parado será resumido, voltando ao estado *running*.
3. Prover uma função própria que será chamada quando este sinal ocorrer através de um mecanismo de captura de sinal.

Um sinal também pode ser disparado programaticamente utilizando chamadas do sistemas através do método *kill*. Deste modo, os sinais podem ser empregados como ferramenta de sincronização ou uma forma primitiva de IPC. No entanto, o processo que dispara e o que recebe

o sinal devem pertencer ao mesmo usuário ou senão o processo emissor deve pertencer ao usuário "root". O método *kill* recebe dois parâmetros, o PID do processo receptor e código do sinal, um número inteiro começando de 1. Esses códigos também são definidos como nomes simbólicos, os quais começam com o prefixo "SIG". Normalmente os nomes simbólicos são utilizados em detrimento aos códigos, uma vez que o código do sinal pode variar entre arquiteturas de hardware. O Linux implementa 31 sinais padrão os quais são sumarizados e descritos no Quadro 3.

2.6 Serialização e Desserialização de Dados

Exceto *Signals*, as demais técnicas de comunicação citadas anteriormente suportam transmissões em formato texto ou binário. Devido a heterogeneidade das linguagens de programação utilizadas neste trabalho, decidiu-se utilizar exclusivamente o formato via texto. Isso faz com que o *kernel* se encarregue em transformar os *bytes* recebidos em caracteres antes de entregá-los à aplicação. Desta forma, o entendimento da comunicação é assegurado desde que emissor e receptor implementem o mesmo protocolo de comunicação. Portanto é necessário que o emissor converta sua estrutura dados em um formato passível de transmissão, no caso texto. Este processo recebe o nome de serialização. A operação reversa, onde o texto é transformado em um estrutura de dados, recebe o nome de desserialização (CARLSON; RICHARDSON, 2015). Vários formatos podem ser utilizados com esta finalidade, como: XML, YAML, JSON, Marshal, Message Pack entre outros.

Quadro 3: Sinais padrão no Linux

Número	Nome	Ação Padrão	Descrição
1	SIGHUP	termina processo	terminal desconectado
2	SIGINT	termina processo	interrompe programa
3	SIGQUIT	termina processo e cria *dump*	sai do programa
4	SIGILL	termina processo e cria *dump*	instrução ilegal
5	SIGTRAP	termina processo e cria *dump*	exceção programada
6	SIGABRT	termina processo e cria *dump*	aborta programa (antigo SIGIOT)
7	SIGEMT	termina processo e cria *dump*	emula uma instrução executada
8	SIGFPE	termina processo e cria *dump*	exceção de ponto flutuante
9	SIGKILL	termina processo	mata programa
10	SIGBUS	termina processo e cria *dump*	erro de barramento
11	SIGSEGV	termina processo e cria *dump*	falha de segmentação
12	SIGSYS	termina processo e cria *dump*	invocação de chamada de sistema inexistente
13	SIGPIPE	termina processo	escrita em pipe sem leitor
14	SIGALRM	termina processo	cronômetro *real-time* expirado
15	SIGTERM	termina processo	sinal de término de *software*
16	SIGURG	descarta sinal	condição urgente presente no *socket*
17	SIGSTOP	para processo	para processo (não pode ser capturado ou ignorado)
18	SIGTSTP	para processo	sinal de parada gerado por teclado
19	SIGCONT	descarta sinal	continue depois de parar
20	SIGCHLD	descarta sinal	processo filho teve estado alterado
21	SIGTTIN	para processo	tentativa de leitura em segundo plano a partir do terminal
22	SIGTTOU	para processo	tentativa de escrita em segundo plano a partir do terminal
23	SIGTTOU	descarta sinal	I/O é possível para um descritor
24	SIGXCPU	termina processo	tempo de CPU excedido
25	SIGXFSZ	termina processo	limite de tamanho de arquivo excedido
26	SIGVTALRM	termina processo	alarme virtual de tempo
27	SIGPROF	termina processo	verificando alarme de tempo
28	SIGWINCH	descarta sinal	janela redimensionada
29	SIGINFO	descarta sinal	*status* requisitado por teclado
30	SIGUSR1	termina processo	sinal definido por usuário
31	SIGUSR2	termina processo	sinal definido por usuário

Fonte: Stevens e Rago (2013). Adaptado.

3 MATERIAIS E MÉTODOS

Normalmente um *cluster* de computadores tem como objetivo primário otimizar ao máximo o tempo de execução de uma aplicação. No entanto, neste trabalho, a facilidade de utilização por parte do usuário final é o objetivo principal, seguido do desempenho computacional. Isso não significa que não se priorizou o desempenho, mas que ao decidir entre duas tecnologias para a mesma finalidade se escolheu a que fosse de mais fácil utilização. Por essa razão, em alguns pontos o desempenho foi sacrificado. No entanto, otimizou-se outras partes da plataforma para mitigar essas perdas. Essa decisão leva em conta o perfil do usuário que a plataforma pretende atender: alunos e professores dos cursos de engenharia que podem não possuir conhecimentos aprofundados em computação paralela distribuída. Logo, optou-se por facilitar a utilização para tornar a plataforma acessível a mais usuários.

Em decorrência disso, separou-se as responsabilidades entre aplicação de gerenciamento do *cluster* e aplicação cliente que é desenvolvida pelo usuário. A primeira é responsável por abstrair as operações distribuídas, definindo uma interface de programação simplificada que oculte as chamadas remotas de métodos. A aplicação cliente consome os serviços oferecidos pelo *cluster*.

Para tornar a plataforma mais flexível, permitindo que um número maior de linguagens de programação fossem compatíveis com o *cluster*, escolheu-se utilizar apenas operações básicas disponíveis na maioria das linguagens: leitura e escrita em terminal. Em virtude disso, as aplicações de gerenciamento e cliente foram implementadas como processos separados. Dessa forma, foram utilizadas técnicas de intercomunicação entre processos para comunicação entre o *cluster* e a aplicação cliente.

Foram utilizados *softwares* gratuitos, computadores e equipamentos de rede cabeada disponíveis. Os laboratórios de informática são de uso compartilhado e já possuem sistemas operacionais e *softwares* utilizados nas aulas. Diante disso, optou-se por carregar um sistema operacional através da rede. Assim é possível utilizar um sistema operacional específico para implementação do *cluster* sem alterar a configuração previamente instalada na máquina.

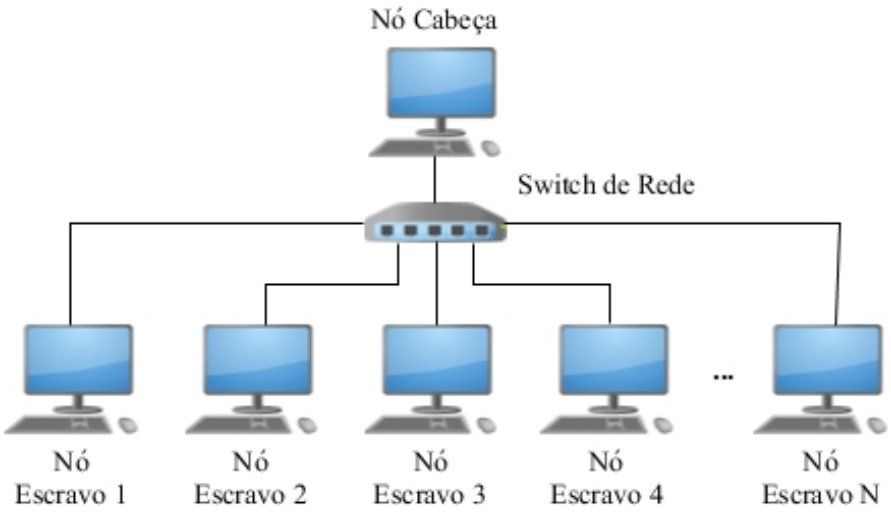
A seguir é apresentada a estrutura do *cluster* proposto neste trabalho.

3.1 Estrutura do *cluster*

Um *cluster* é composto por um conjunto de máquinas que executam uma aplicação distribuída de forma colaborativa. Normalmente um desses computadores é responsável exclusivamente por coordenar os demais. Nessa estrutura, o computador responsável pelo gerenciamento é classificado como Nó Cabeça, enquanto os demais são chamados Nós Escravos. A Figura 7 ilustra a estrutura geral de um *cluster*. Como pode ser observado, todos os computadores estão interconectados por uma rede.

Para que o ambiente distribuído seja transparente para aplicação cliente, é necessário que o *cluster* possua uma arquitetura capaz de gerenciar as rotinas de comunicação. Dessa forma, é possível paralelizar a aplicação cliente como se fosse executada em um único computador. Para tanto, neste trabalho desenvolveu-se a arquitetura apresentada na Figura 8. Ela está dividida

Figura 7: Estrutura física de um *cluster* genérico.



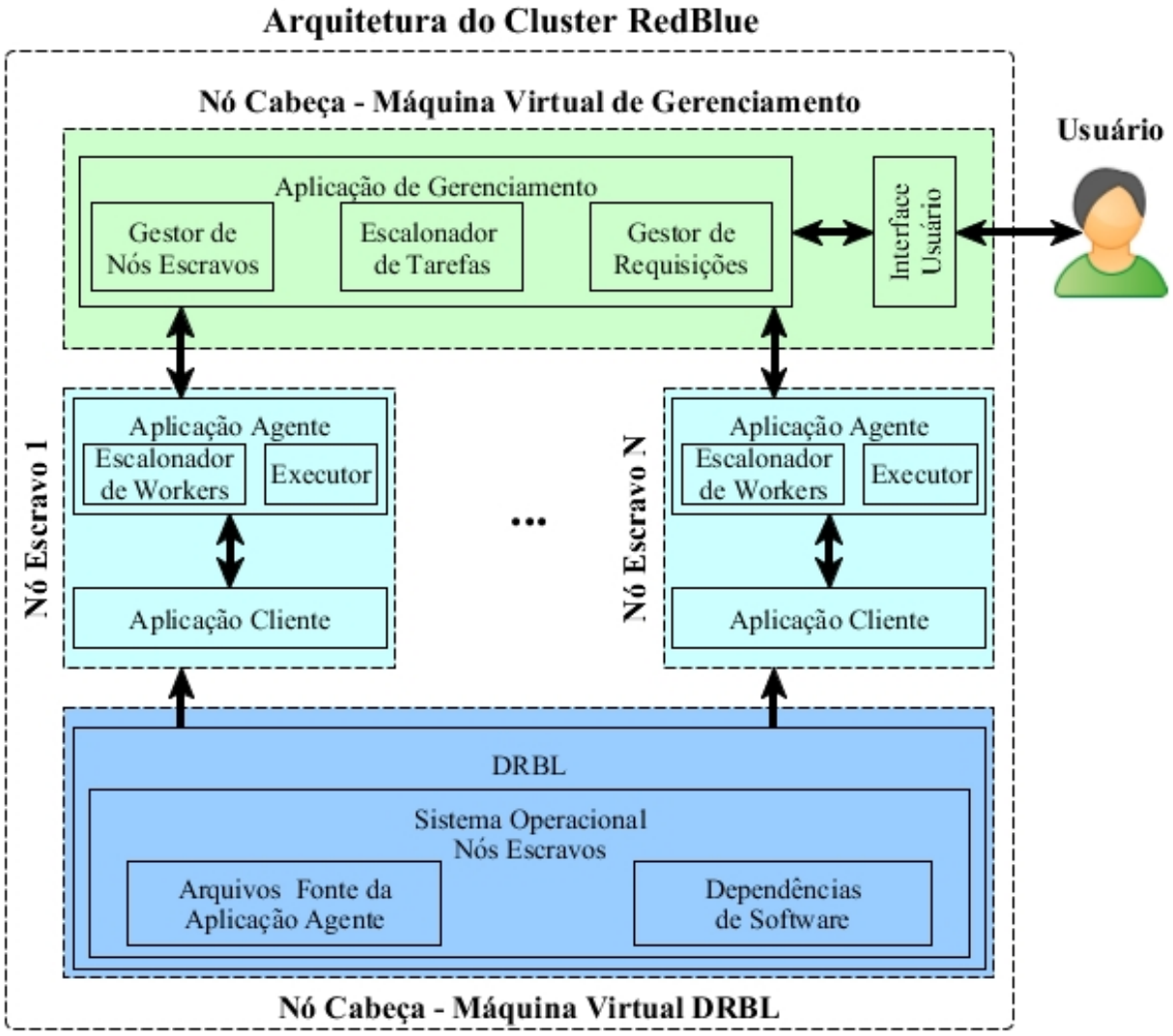
Fonte: Próprio autor.

em três partes principais: Nó Cabeça, responsável por receber as requisições do usuário, gerenciar os computadores do *cluster* e coordenar a distribuição de tarefas; Nós Escravos, encarregados de executar e monitorar a aplicação cliente, bem como capturar os resultados e enviá-los ao Nó Cabeça; e DRBL (*Diskless Remote Boot in Linux*) (SHIAU *et al.*, 2017), incumbido de disponibilizar uma imagem do sistema operacional, previamente configurado com as dependências de *software* necessárias à aplicação cliente, para os nós escravos.

O Nó Cabeça e o DRBL são instalados em máquinas virtualizadas distintas em um mesmo computador. Optou-se pela virtualização por três razões. A primeira para que se pudesse utilizar uma única máquina física para gerenciamento do *cluster* e disponibilização da imagem do sistema na rede. A segunda porque o *software* DRBL modifica muitas funcionalidades do sistema operacional, uma vez que instala muitos pacotes, edita regras de *firewall* e cria compartilhamentos via NFS (*Network File System*) entre outros, podendo comprometer o funcionamento do sistema do usuário. A terceira para que se pudesse manter um *template* do sistema operacional pré-configurado para utilização do *cluster*. Basta clonar a máquina virtual DRBL e adicionar as dependências de pacotes da aplicação cliente. Assim é possível manter uma coleção de máquinas virtuais (*Virtual Machines* - VMs) configuradas com dependências de *software* distintas. Dessa forma, para utilizar uma aplicação diferente, basta ligar uma outra VM DRBL previamente configurada.

A comunicação entre Aplicação de Gerenciamento e Aplicações Agentes é realizada por meio de chamadas de métodos remotos, assim como a comunicação entre Interface Usuário e Aplicação de Gerenciamento. Entretanto, os arquivos executáveis da Aplicação Cliente são disponibilizados via NFS. O Nó Cabeça disponibiliza uma pasta compartilhada onde os arquivos do usuário são copiados e mapeados para os Nós Escravos em modo *somente leitura*.

Figura 8: Arquitetura do *cluster*.



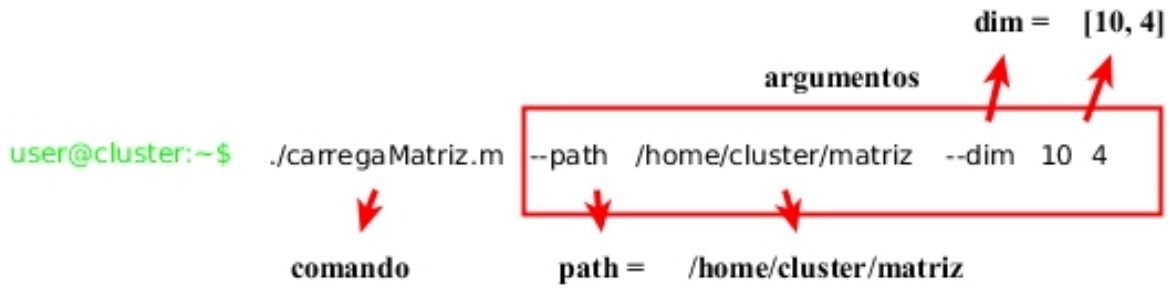
Fonte: Próprio autor.

3.1.1 Formato de comunicação

Para permitir que o usuário escolha a linguagem de desenvolvimento da aplicação, optou-se por separar o gerenciamento do *cluster* e a Aplicação Cliente em nível de processos. Desta forma, o *cluster* se responsabiliza por executar a Aplicação Cliente passando os parâmetros necessários para sua inicialização e por recuperar os dados de saída dessa aplicação. O cliente é responsável por retornar os dados em uma estrutura pré-estabelecida, incluindo instruções de controle.

Para o formato de entrada de opções e argumentos da Aplicação Cliente, utiliza-se um mecanismo similar à passagem de argumentos na linha de comandos do Linux. Um exemplo de utilização é apresentado na Figura 9. Um parâmetro é definido utilizando-se dois traços (--) seguidos do seu nome e uma lista de valores separados por espaços. Escolheu-se essa notação para manter compatibilidade com aplicações via terminal. Isso facilita o desenvolvimento por

Figura 9: Exemplo de passagem de parâmetros via terminal.



Fonte: Próprio autor.

permitir que as aplicações executem da mesma forma tanto no terminal quanto no *cluster*, o que torna possível testar uma tarefa individualmente no terminal.

A comunicação entre *cluster* e Aplicação Cliente utiliza o formato JSON *(Javascript Object Notation)* (ECMA-404, 2013) e um conjunto de chaves reservadas para controle do *cluster*. Essa notação é independente de linguagem e de fácil compreensão por humanos e máquinas, o que facilita a geração e manipulação de dados. Além disso, existe um vasta gama de bibliotecas que auxiliam a manipulação da dados no formato JSON (LEPILLEUR, 2010; Python Software Foundation, 2017; COUTURE-BEIL, 2014; FANG, 2017). Os resultados obtidos da execução das aplicações clientes e o conjunto de tarefas iniciais submetidas ao *cluster* são codificadas no formato JSON respeitando a estrutura de controle. A lista a seguir exibe as palavras reservadas e a semântica utilizada pela estrutura de controle do *cluster*:

- *command* - Recebe o nome do arquivo que será executado incluindo sua extensão, quando for o caso. Por exemplo, caso o arquivo possua o nome *job_setup.m*, o campo *command* receberá o valor "job_setup.m".
- *args* - Recebe uma *string* contendo os parâmetros para aplicação definida em *command*. O formato dessa *string* segue o padrão "*--param1 argumento1 argumento2 --param2 argumento3*". Esses parâmetros são passados à aplicação cliente conforme é exibido na Figura 9.
- *wait* - Recebe apenas os valores em formato texto *"true"* ou *"false"*. É utilizado pelo escalonador do *cluster* para definir a ordem de execução das tarefas da aplicação cliente. No conjunto inicial de tarefas e entre os elementos definidos no campo *next_jobs*, apenas uma tarefa pode ter o campo *wait* definido como *"true"*. Este tópico será abordado com mais detalhes na Seção 3.1.2.
- *job_output* - Recebe os resultados da execução da aplicação em formato JSON. A definição dos nomes das chaves e valores são de responsabilidade do usuário. Os valores definidos nesse campo serão redirecionados à tarefa responsável pelo agrupamento de informações ou serão a saída final da execução do conjunto de tarefas.

- *next_jobs* - Recebe um *array* de objetos em formato JSON contendo às chaves *command*, *args*, e *wait* quando deseja-se encadear novas tarefas ou um *array* vazio quando a tarefa não possui descendentes. As chaves *command*, *args* e *wait* possuem as mesmas funcionalidades já destacadas anteriormente e seguem a mesma estrutura da definição inicial de tarefas. Por exemplo, caso *next_jobs* esteja definido como {*"next_jobs": []*}, não serão criadas novas tarefas. No entanto, uma definição conforme:

  {*"next_jobs": [{"command": "job1.m", "args": "--valor 1", "wait":"true"},{"command": "job2.m", "args": "--opcao a", "wait":"false"}]*}

  indica que serão criadas as tarefas *job1.m* e *job2.m*, de forma que *job1.m* será executado apenas após o término de *job2.m*. A precedência entre tarefas será abordada na Seção 3.1.2.
- *job_input* - Recebe um *array* em formato JSON com os valores de *job_output* definidos por outras tarefas. Esse campo é criado pelo *cluster* e utilizado para repassar as informações contidas no campo *job_output* de outras tarefas à responsável por agrupar esses resultados. Dessa forma, o *cluster* adiciona o parâmetro *job_input* com seus respectivos valores à lista de parâmetros definida em *args*. Este *array* é ordenado seguindo sequencia relativa em que as tarefas foram criadas no campo *next_jobs*. Por exemplo, se a tarefa "c" é responsável por agrupar os valores das tarefas "a" e "b" com saídas respectivamente iguais a {"val": "1"} e {"val": "2"}, a tarefa "c" receberá o parâmetro *job_input* da seguinte forma:

  --job_input "[{\"val\": \"1\"}, {\"val\": \"2\"}]".

  Ressalta-se que é necessário a utilização da barra de escape (\) antes das aspas que envolvem parâmetros e valores, uma vez que às aspas são delimitadores de *strings*. No entanto, o *cluster* se encarrega deste processo automaticamente.

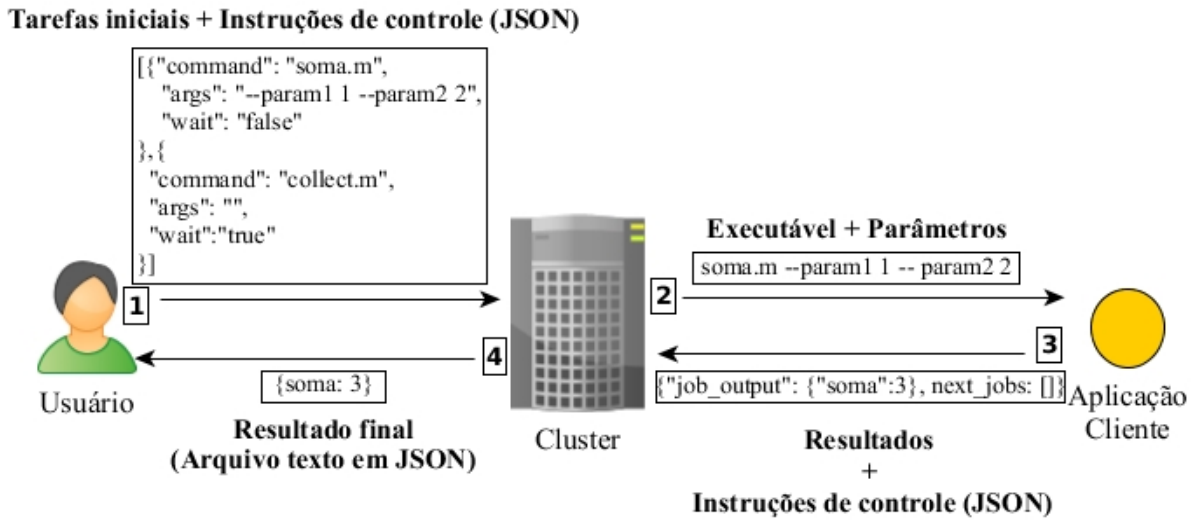
O *cluster* retorna o resultado final da execução da Aplicação Cliente ao usuário por meio de um arquivo de texto codificado em JSON. Os dados contidos nesse arquivo são provenientes do campo *job_output*, extraído do retorno da instância final da Aplicação Cliente.

A Figura 10 resume o fluxo de dados e os formatos utilizados na comunicação entre usuário, *cluster* e Aplicação Cliente. Também é possível combinar o formato de passagem de parâmetros via terminal com a notação JSON para representar estruturas de dados mais complexas. Basta definir o valor de um parâmetro como uma *string* no formato JSON. Essa é a mesma estratégia utilizada pelo *cluster* para combinar os valores de *job_output* e submetê-los à aplicação utilizando o parâmetro *job_input*.

3.1.2 Escalonador de tarefas

Para que uma aplicação cliente obtenha vantagem ao ser submetida ao *cluster* é necessário que essa possa ser dividida em partes menores capazes de executar em paralelo. Essas partes menores recebem o nome de *jobs*, tarefas. No entanto, é comum que uma aplicação possua partes paralelizáveis enquanto outras são inerentemente sequenciais. Normalmente um trecho de código sequencial surge após um paralelo, atuando como ponto de sincronização ou

Figura 10: Formato e fluxo de informações no *cluter*.



Fonte: Próprio autor.

barreira. Isso é utilizado para garantir que o fluxo do programa continue apenas após o término de todas as operações paralelas anteriores, garantindo que todas as informações necessárias estejam disponíveis para a próxima operação.

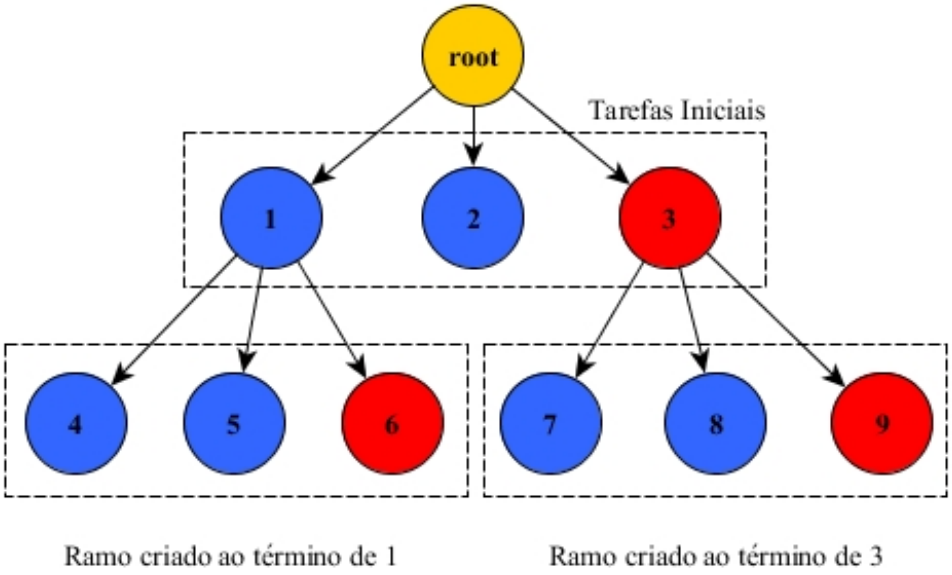
Dessa forma, faz-se necessário um mecanismo pelo qual o usuário possa informar ao *cluster* quais tarefas podem ser executadas em paralelo, qual a ordem de execução das tarefas e quais dessas funcionarão como pontos de sincronização. Para isto, organizou-se às tarefas em uma estrutura de dados em forma de árvore conforme à Figura 11, chamada Árvore de Tarefas *(JobsTree)* (RAUBER; RÜNGER, 2010).

Nessa estrutura, seus elementos recebem o nome de nós (nesta seção, a palavra nó se refere a um elemento da Árvore de Tarefas e não a um computador). Os nós são organizados hierarquicamente por meio de arestas. Cada nó possui um único nó superior, chamado nó pai, com exceção do nó raiz (*root*). Os descendentes diretos de um nó chamam-se nós filhos. Um nó que não possui descendentes recebe o nome de nó folha. Nós que possuem pai comum chamam-se irmãos. Um conjunto de nós com pai comum recebe o nome de ramo.

Cada nó representa uma tarefa, com exceção do nó raiz. Este nó especial é utilizado para indicar o ponto de entrada da árvore, logo não armazena dados referentes às tarefas. As tarefas iniciais são adicionadas a esse nó. A medida em que as tarefas terminam, elas podem adicionar novas tarefas, de forma que a árvore é definida dinamicamente.

Em cada ramo deve existir obrigatoriamente um nó que atuará como ponto de sincronização. Este é identificado pelo campo *wait* com valor *"true"*, enquanto os demais irmãos recebem o valor *"false"*. Um nó que atua como ponto de sincronização é classificado como *RedNode*, representado em vermelho, enquanto os demais são classificados como *BlueNode*, representado em azul.

Figura 11: Árvore de Tarefas.



Fonte: Próprio autor.

Um *RedNode* será executado apenas após seus irmãos *BlueNode*, e os descendentes deste, terminarem a execução. Os *BlueNodes*, por sua vez, não necessitam verificar o estado da execução dos nós irmãos, de forma que podem executar assim que houver computadores aptos a recebê-los. Por exemplo, considerando a Figura 11, a tarefa contida no nó 3 será executada apenas após o término das tarefas dos nós 1, 2, 4, 5 e 6. Neste caso, os nós 2, 4 e 5 poderiam executar em paralelo. Da mesma forma, a tarefa do nó 6 será executada apenas após o término das dos nós 4 e 5. Já a tarefa condida em 9 será executada após o término de 7 e 8. Essas regras são utilizadas para definir a ordem relativa de execução das tarefas. Entretanto, não há como saber a ordem exata de execução, visto que isso depende do tempo gasto na execução de cada tarefa e da disponibilidade de computadores para executá-las.

Como os *RedNodes* atuam como ponto de sincronização é necessário que eles tenham acesso aos resultados produzidos pelos seus nós irmãos e os descendentes destes. Uma forma de se conseguir isso é por meio de regras de escalonamento, como as regras a seguir:

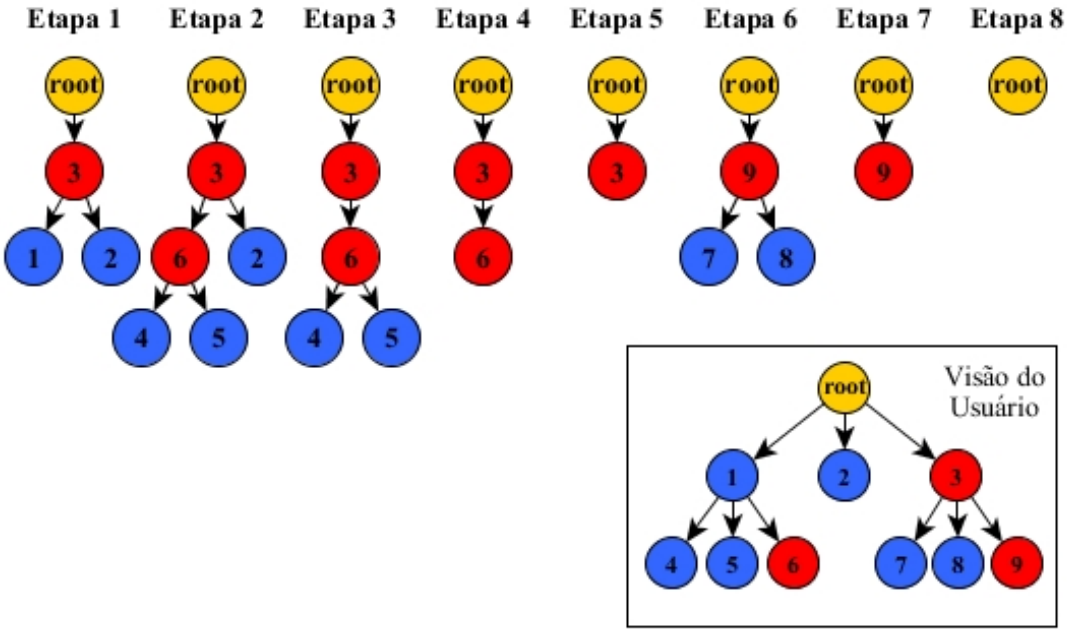
1. Um nó que possui filhos deve repassar seus resultados a eles.
2. *BlueNodes* sem filhos devem repassar seus resultados ao irmão *RedNode*.
3. *RedNode* sem filhos devem repassar seus resultados ao próximo ancestral *RedNodes* ainda não executado.
4. Caso nenhuma das regras anteriores sejam aplicáveis, terminar a aplicação.

Essas regras seriam necessárias porque a estrutura da árvore reflete apenas a ordem em que as tarefas foram inseridas, fazendo com que o fluxo da informação e à ordem de execução sejam diferentes da hierarquia definida pelas arestas. Isso implicaria em uma busca da raiz até às folhas a cada término de tarefa para verificar quais nós estão aptos para execução e outra para

definir qual nó receberia a informação. Além disso, os resultados das tarefas seriam mantidos nos nós, que apenas seriam marcados como executados. Isso tornaria as operações mais lentas devido ao crescimento da Árvore de Tarefas.

Uma solução para esses problemas é rearranjar as arestas da árvore para que essas representem o fluxo da informação. Isso pode ser feito promovendo o nó RedNode de cada novo ramo a pai de seus irmãos. Como consequência, nós sem descendentes sempre repassarão sua informação ao nó pai. Além disso, um nó recém terminado é substituído pelos seus descendentes. Dessa forma, mantêm-se na árvore apenas aquelas tarefas que estão aptas a executar.

Figura 12: Árvore de Tarefas do ponto de vista do escalonador.



Fonte: Próprio autor.

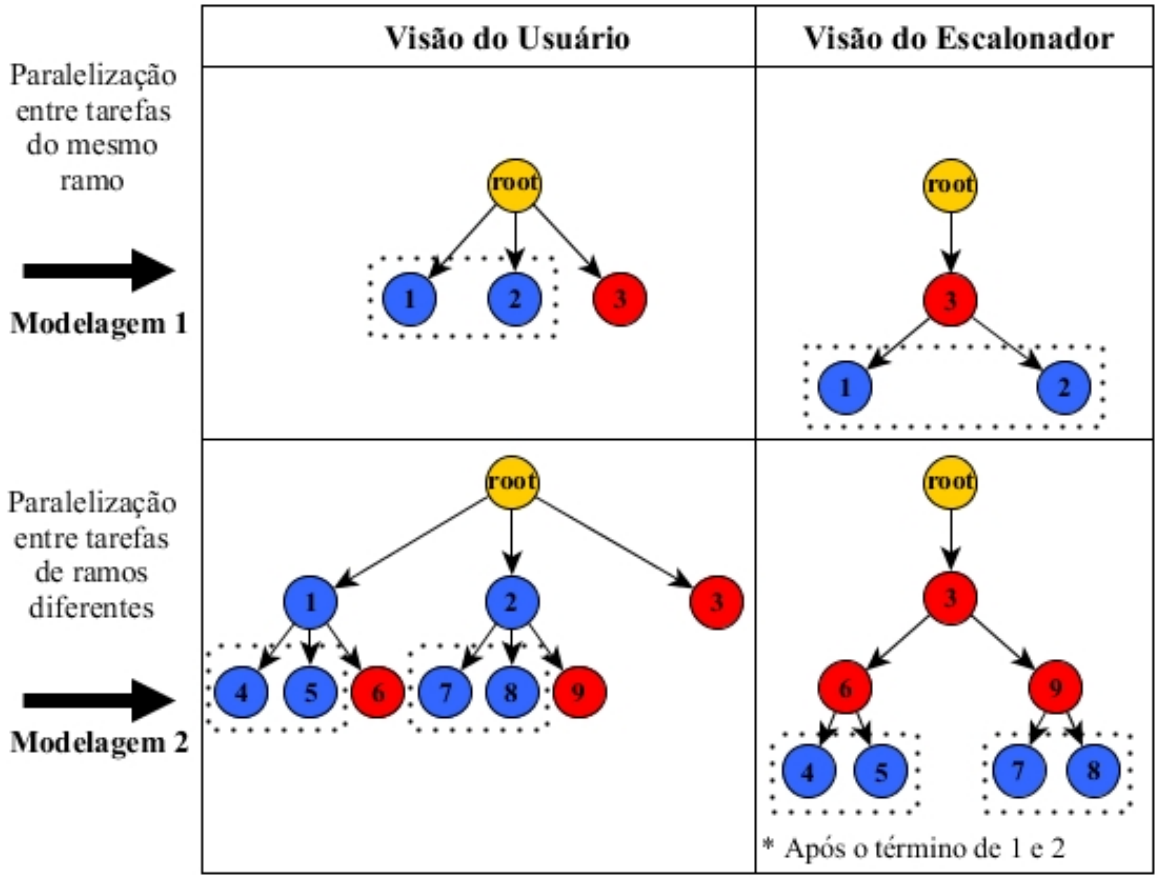
O usuário continua a definir às tarefas iniciais e as novas da mesma forma, mantendo um único *RedNode* a cada ramo. Para o usuário, a visão da Árvore de Tarefas permanece inalterada, uma vez que as arestas representam o parentesco entre tarefas. No entanto, o *cluster* redefine internamente a estrutura da árvore para simplificar as operações, de forma que o nó pai sempre receba o resultado da execução de tarefas que não criem novos descendentes. Essa estrutura faz com que as tarefas contidas nos nós folhas sejam as que estão aptas a executar, uma vez que essas não possuem tarefas dependentes. Por exemplo, na Etapa 1, apenas os nós 1 e 2 estão aptos a executar, uma vez que não possuem descendentes, ou seja, são folhas. Na Etapa 2, a tarefa contida no nó 1 terminou e suas informações foram repassadas aos nós 4, 5 e 6 no momento da definição do novo ramo. Internamente o *cluster* rearranjou esse ramo, promovendo 6 a pai de 4 e 5. Logo, quando esses terminaram, na Etapa 3, o resultado será enviado a 6. O mesmo ocorre na Etapa 4, 6 é um nó folha e por isso está apto a executar. Ao terminar sua

informação é repassada ao seu pai, o nó 3. A condição de parada passa a ser a inexistência de nós na Árvore de Tarefas.

Nessa estrutura também é assegurado que as informações repassadas aos *RedNodes* estarão na mesma ordem em que as tarefas filhas foram definidas. Por exemplo, ao observar a Figura 12, na Etapa 6, verifica-se que no momento da definição das novas tarefas, o nó 7 estava a esquerda do nó 8. Logo, a primeira posição do campo *job_input* da tarefa do nó 9 receberá o resultado da tarefa do nó 7 e a segunda posição o resultado da tarefa do nó 8. Isto é utilizado para se identificar a saída de operações diferentes que executam em paralelo no mesmo ramo.

O escalonador também permite que o usuário paralelize a execução de conjunto de tarefas, o que ocorre quando nós *BlueNodes* criam descendentes. Neste caso, além da execução paralela entre *BlueNodes* do mesmo ramo, acorrerá também a execução paralela entre ramos diferentes. Isto é útil quando deseja-se sub-dividir uma rotina já paralelizada em operações menores que também são paralelizáveis. Essa abordagem é ilustrada na Figura 13.

Figura 13: Paralelização de tarefas em um mesmo ramo e em ramos diferentes.



Fonte: Próprio autor.

Na Figura 13, os retângulos pontilhados representam um conjunto de tarefas paralelizáveis na visão do usuário e do escalonador de tarefas. A Modelagem 1 apresenta uma abordagem monolítica, de modo que a rotina paralelizável de mais alto nível foi implementada

em uma única tarefa. Neste caso, apenas duas tarefas poderão ser executadas no *cluster* ao mesmo tempo, respectivamente as tarefas 1 e 2. Já a Modelagem 2 apresenta o desmembramento das tarefas 1 e 2 em outras menores, fazendo com que suas rotinas possam ser executadas em até quatro unidades de processamento simultaneamente, respectivamente as tarefas 4, 5, 7 e 8. O aumento no nível de paralelismo pode melhorar o desempenho da aplicação desde de que as tarefas resultantes do desmembramento permaneçam com elevado tempo de execução, superando o *overhead* causado pela criação das tarefas adicionais.

3.1.3 Atribuição de tarefas e balanceamento de carga

Os processadores atuais são dotados de múltiplos núcleos. Deste modo, atribuir uma única tarefa a um computador pode significar desperdício de recursos dependendo da natureza do problema. Por exemplo, pode-se criar múltiplas instâncias de uma aplicação *single thread* com alto consumo de CPU e baixo consumo de memória para aproveitar todo o potencial do processador, com baixo risco de escassez de recursos. Porém, caso a aplicação consuma muita memória ou utilize vários núcleos de processamento, a criação de novas instâncias causará deterioração do desempenho, uma vez que acirrará a disputa por recursos computacionais.

Para tornar possível que o usuário escolha a melhor alternativa para sua aplicação, optou-se por dividir cada computador (Nó Escravo) em estruturas menores chamadas *Workers* (Trabalhadores). O número de *Workers* determina quantas tarefas um computador pode alocar simultaneamente. Por padrão, o número de *Workers* é determinado pela quantidade de núcleos físicos disponíveis em cada computador. No entanto, também pode ser configurado como a quantidade de núcleos lógicos ou um número fixo definido pelo usuário. Neste último caso, o número de *Workers* não depende da quantidade de núcleos físicos ou lógicos, de modo que pode ser atribuído um valor maior ou menor do que o número de núcleos. A Seção 3.1.5 trata com mais detalhes o mecanismo interno de divisão de tarefas de um computador.

Mesmo com a capacidade multitarefa dos processadores atuais, aprimorada pelo uso de múltiplos núcleos, ela é limitada aos recursos de um computador. Dessa forma, o *cluster* deve distribuir as tarefas de modo a não sobrecarregar um computador enquanto outros estão subutilizados. Para isso, o Nó Cabeça deve conhecer os recursos disponíveis no *cluster*, descobertos durante a fase de inicialização de serviços.

Ao iniciar, cada computador informa ao Gestor de Nós Escravos seu endereço de rede e o número de *Workers* disponíveis. Em seguida, o Nó Escravo ficará em estado de espera, aguardando a requisição do Escalonador de Tarefas. Então, esse gestor organiza as informações dos Nós Escravos em uma lista circular chamada *nodes queue*, adicionando a cada posição os campos *busy* e *free*. Esses campos representam a quantidade de *Workers* alocados e livres, respectivamente. Dessa forma, utilizando-se computadores *quad core*, por exemplo, cada um iniciará com 4 *Workers* livres. Após essas operações, o Nó Cabeça estará pronto para receber as requisições do usuário e iniciar a distribuição de tarefas.

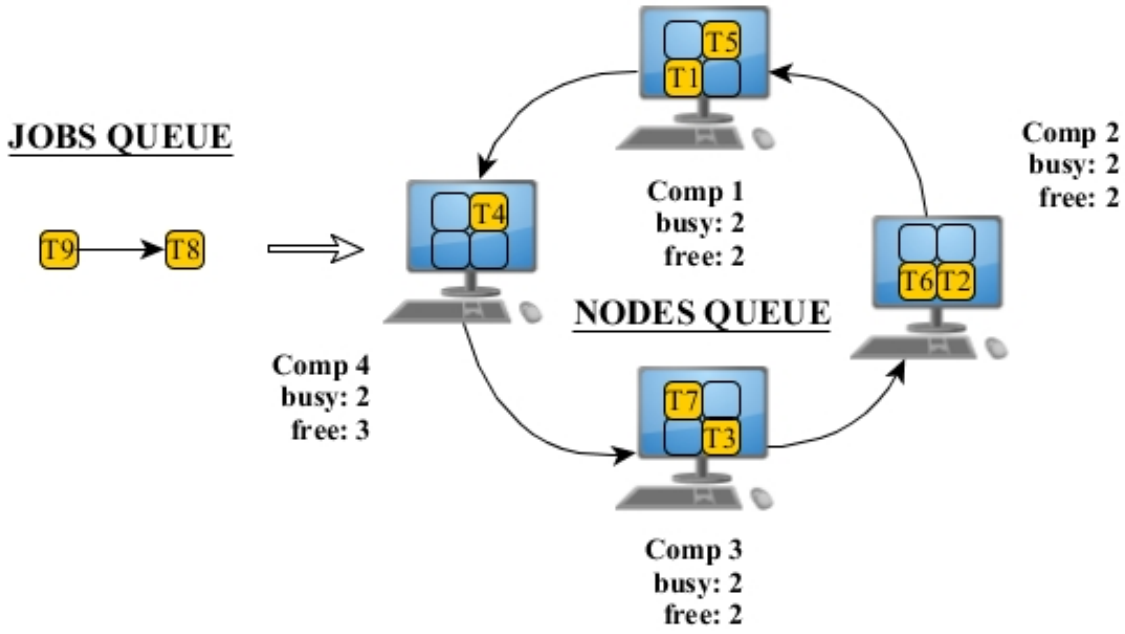
Ao receber as tarefas iniciais, o Gestor de Requisições solicita ao Escalonador de Tarefas que monte a Árvore de Tarefas. Como visto na Seção 3.1.2, apenas tarefas aptas ou em

execução são mantidas na árvore. Logo, é necessário saber o estado das tarefas para não correr o risco de executá-las mais de uma vez. Por essa razão, cada tarefa armazena seu estado em um campo chamado *status*. A Seção 3.1.4 detalha os estados que uma tarefa pode assumir.

Durante a inicialização da árvore, o estado das tarefas é definido como *stopped* (parada). Das folhas da árvore gera-se uma lista de tarefas aptas a executar. Dessa, verifica-se o campo *status* de cada tarefa, mantendo-se apenas as que possuem o *status stopped*. São criadas referências às tarefas remanescentes, que são então inseridas no fim da fila de espera para execução, chamada *jobs queue*. Assim que as novas referências de tarefas são inseridas no fim dessa fila, o estado delas muda para *queued* (enfileirada). Isso garante que essas tarefas não sejam novamente incluídas na fila de execução durante o processo de avaliação da árvore. Esse mecanismo permite definir dinamicamente uma fila de execução que respeita a precedência entre as tarefas. Dessa forma, a próxima tarefa a ser executada será sempre a primeira na fila de execução.

Após atualizar essa fila, o Escalonador de Tarefas solicita ao Gestor de Nós Escravos um computador disponível para executar a próxima tarefa, situação que é ilustrada na Figura 14. Este checa o somatório total de *Workers* disponíveis e, caso seja maior que zero, verifica o primeiro computador da lista de computadores (*nodes queue*).

Figura 14: Mecanismo Atribuição de tarefas.



Fonte: Próprio autor.

Caso esse computador possua *Workers* livres, a primeira tarefa da lista *jobs queue* será submetida a ele. Então a referência à tarefa será removida da fila de espera para execução (*jobs queue*) e terá seu *status* mudado para *running*. O computador, por sua vez, terá o número de *Workers* recalculado e será movido para o fim da fila de computadores (*nodes queue*).

Caso verifique-se que o computador não possua *Workers* disponíveis, este também será movido para o fim da fila e o próximo será avaliado. Após completar o ciclo na lista de computadores, esses serão reordenados de forma decrescente pelo número de *Workers* livres. Essa estratégia, juntamente com o giro da lista de computadores, busca manter a carga de tarefas balanceadas entre os computadores do *cluster*.

Ao término de uma tarefa, o computador que a executou tem o número de *Workers* disponíveis incrementado. Caso a tarefa defina novos descendentes, essas informações serão utilizadas para atualizar a Árvore de Tarefas e o ciclo recomeçará até que não restem mais tarefas a serem executadas.

Ressalta-se que a fila de execução (Escalonador de Tarefas) e a lista circular de computadores (Gestor de Nós Escravos) foram propositalmente construídas de forma separada para permitir que sejam independentes da estrutura do *cluster*. Logo o *cluster* é capaz de se adequar a quantidade de computadores utilizados sem que seja necessário mudar qualquer configuração.

3.1.4 Tratamento de erros e estados de uma tarefa

Como mostrado na Seção 3.1.3, uma tarefa assume vários estados durante o ciclo de vida no *cluster*. Esses estados são utilizados para determinar em qual fase de processamento uma tarefa está e também para que seja possível revertê-la em caso de falhas. O Quadro 4 resume o significado de cada estado possível.

O estado da tarefa determina a ação a ser tomada em caso de falha na Aplicação Cliente. Quando um erro ocorre em uma tarefa, os dados do *backtrace* (pilha de chamada de funções) são exibidos para o usuário e a tarefa recebe o estado de erro. Caso a aplicação esteja com outras tarefas em execução, essas serão identificadas e forçadamente encerradas pelo *cluster*. Isto é possível porque o Escalonador de Tarefas armazena o endereço de quem está executando a tarefa e o PID (*Process Identifier*) do processo. Essas informações são enviadas pela Aplicação Agente quando inicia-se a execução da Aplicação Cliente. As tarefas forçadamente encerradas são marcadas como *aborted*. As tarefas que estavam na fila de execução, mas não chegaram a executar, retornam ao estado *stopped*.

Caso a solução do problema esteja contida na própria tarefa que ocasionou o erro, é possível a corrigir e resumir a execução a partir daquele ponto. Isto porque o Escalonador de Tarefas armazena em memória a Árvore de Tarefas com erro e segue para a requisição seguinte. Após corrigir o problema o usuário pode requisitar ao Gestor de Requisições que continue a executar a árvore anterior, bastando informar o identificador da requisição com erro. No entanto, se o erro foi causado por um valor incorreto de uma tarefa anterior, não será possível resumir a aplicação, visto que a tarefa que causou o erro não está na Árvore de Tarefas. Neste caso, o usuário terá que fazer uma nova requisição ao *cluster*.

3.1.5 Workers: Implementação e mecanismo de Alocação de Tarefas

Embora o Gestor de Nós Escravos conheça o quantitativo de *Workers* livres e ocupados em cada computador, esse não atribui a tarefa diretamente a um *Worker* e sim ao

Quadro 4: Estados de uma Tarefa

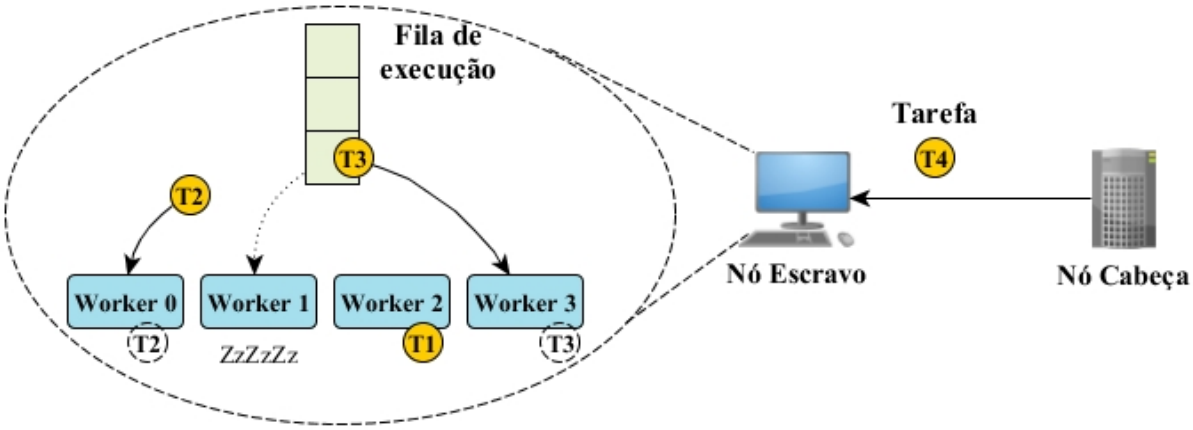
Estado	Descrição	Localização
stopped	Aguardando a execução de tarefas dependentes. Tarefa recém adicionada.	**Nó cabeça**. Exclusivamente na Árvore de Tarefas.
queued	Tarefa movida para fila de execução. Aguarda um Worker disponível.	**Nó cabeça.** Na Árvore de Tarefas e na fila de execução (jobs queue).
running	Tarefa em execução. Processo criado em algum nó escravo.	**Nó cabeça.** Na Árvore de Tarefas. **Nó escravo.** Em execução em um Worker
finished	Tarefa terminada com sucesso.	**Nó cabeça.** Apenas em *logs* de execução
error	Erro de execução da aplicação cliente. Tarefa com erro.	**Nó cabeça.** Na Árvore de Processos.
aborted	Terminada forçadamente pelo *cluster*.	**Nó cabeça.** Na Árvore de Processos, pode estar na fila de execução. **Nó escravo.** Ainda pode estar em execução em um Worker

Fonte: Próprio autor.

Nó Escravo. Optou-se por essa estratégia para simplificar a implementação e diminuir o tempo gasto pelo Nó Cabeça para atribuição de tarefas. Logo, essa responsabilidade é delegada a Aplicação Agente assim que esta recebe a tarefa, deixando o Nó Cabeça disponível para a próxima requisição. Isso é possível porque cada agente implementa seu mecanismo de distribuição de tarefas, chamado Escalonador de *Workers*, exibido na Figura 15.

Esse foi desenvolvido utilizando-se a técnica *Master-Worker*, onde um processo é responsável por coordenar as atividades dos demais (RAUBER; RÜNGER, 2010). A Aplicação Agente cria uma fila de tarefas e instancia um servidor local para que os processos filhos (*Workers*) a acessem. Então essa cria *N* novos processos por meio da operação de cópia (*fork*), onde *N* é o número máximo de tarefas que podem ser alocadas simultaneamente, definido por parâmetro. Os processos filhos são programados para continuamente buscar e executar as tarefas da fila de execução por meio do servidor local. Este garante que apenas um processo terá acesso a uma tarefa de cada vez. Quando o processo tenta acessar a fila e não resta nenhuma tarefa, recebe o sinal “STOP”, entrando em estado de dormência. Quando uma requisição de processamento de tarefa chega na Aplicação Agente, essa o enfileira e dispara o sinal “CONT”, despertando os processos adormecidos, que buscam novamente na fila, reiniciando o ciclo. Isso

Figura 15: Mecanismo de alocação de tarefas entre *Workers*.



Fonte: Próprio autor.

consequentemente garante o balanceamento de carga, uma vez que apenas *Workers* ociosos buscarão uma tarefa na fila.

Ainda é possível associar um *Worker* a um núcleo de processamento configurando o parâmetro do *cluster* CPU_AFFINITY para "yes". Isso faz com que tarefas submetidas a um *Worker* executem em um núcleo predefinido. Isto é útil para aplicações *single thread*, pois aumenta a probabilidade de que os dados recém utilizados estejam no *cache* do processador, melhorando o desempenho (LOVE, 2013). Além disso, verificou-se que o escalonador de processos padrão do sistema não faz considerações sobre o balanceamento de carga entre núcleos lógicos pertencentes ao mesmo núcleo físico. Isso fez com que, em diversos casos, fossem alocadas duas tarefas sobre um mesmo núcleo físico, mesmo havendo outros núcleos ociosos. A alocação estática utilizada pelo *cluster* trata esse tipo de problema promovendo uma alocação mais eficiente para aplicações *single thread*. No entanto, essa opção é desativada por padrão, pois pode piorar o desempenho de aplicações *multithread*, cabendo ao usuário ativá-la quando oportuno.

O Escalonador de *Workers* também permite que sejam submetidas mais tarefas do que o número de *Workers*, armazenando-as temporariamente. Embora o Escalonador de Tarefas não submeta mais tarefas que o número de *Workers* ociosos, na implementação atual, essa capacidade poderia ser utilizada para diminuir a latência entre às execuções de tarefas. Uma vez que com a próxima tarefa enfileirada localmente, esta poderia ser iniciada enquanto, paralelamente, os resultados da anterior são enviados ao Gestor de Requisições. Pretende-se adicionar essa funcionalidade nas próximas versões do *cluster* deixando-a como opção do usuário ativá-la ou não.

3.1.6 Execução de código cliente e meio de comunicação

A execução do código cliente é similar a inicialização de uma aplicação no terminal do Linux. Normalmente, aplicações via linha de comando possuem como saída padrão (STDOUT) o próprio terminal, e recebem dados via entradas de texto também pelo terminal,

sendo essa a entrada padrão (STDIN). Neste caso, é comum que as exceções sejam exibidas na tela (STDERR). A diferença básica para execução no *cluster* é que os dados de entrada e saída serão manipulados por outro processo (*Worker*), que terá acesso a essas informações ao redirecionar os descritores padrão. Isso é realizado por meio da criação de *pipes* associados à operação *fork* (MATTHEW; STONES, 2009).

Antes de iniciar a Aplicação Cliente, o Executor define três pares de *pipes* que serão conectados respectivamente a STDIN, STDOUT e STDERR. Cada *Worker* possui internamente seu Executor. Como visto anteriormente, um *Worker* é um processo gerenciado pela Aplicação Agente.

Para iniciar a Aplicação Cliente, o Executor cria uma cópia de seu próprio processo por meio de uma operação *fork*. O processo filho então chama a função *exec* passando como argumento o caminho do executável da Aplicação Cliente. Neste momento, o processo filho, clone do *Worker*, passa a ser a própria Aplicação Cliente. No entanto, compartilha os descritores de arquivos (*pipes*), criados pelo processo pai. Isto faz com que o processo *Worker*, por meio do Executor, possua um canal de comunicação comum, permitindo a troca de informações com a Aplicação Cliente.

A princípio utilizou-se apenas os descritores STDOUT e STDERR, respectivamente para captura das mensagens de saída e erro da Aplicação Cliente. A entrada de dados era realizada por meio de outro mecanismo, o vetor de argumentos (ARGV) (MATTHEW; STONES, 2009). Este divide automaticamente o texto inserido após o executável da aplicação utilizando os espaços como separador. Isto permitia que a Aplicação Cliente utilizasse o utilitário de recuperação de parâmetros, comumente conhecido como *option parser* (COPELAND, 2013), para recuperar os parâmetros no *cluster* e terminal de maneira idêntica. O formato da passagem de parâmetros foi apresentado na Seção 3.1.1.

No entanto, verificou-se que o vetor de argumentos possui um tamanho máximo definido. Assim, caso a lista de argumentos ultrapasse esse valor, uma exceção será disparada pelo sistema operacional. Por essa razão, passou-se a utilizar a passagem de parâmetros via STDIN *(Standart Input)*. Isto porque embora os *pipes* utilizem um pequeno espaço de armazenamento temporário (*buffer*), estes podem transmitir qualquer quantidade de informação. Para isso, utiliza-se várias transmissões consecutivas, tornando possível o envio de grande quantidade de dados.

Mesmo trocando o meio de entrada de dados, manteve-se o comportamento do vetor de argumentos. Desse modo, o utilitário para extração dos parâmetros funcionará para STDIN ou ARGV, sendo o primeira opção utilizada no *cluster*. Para testar uma Aplicação Cliente via terminal usando ARGV, basta incluir o argumento *--argv* em sua lista de parâmetros.

A saída de dados é obtida por meio da captura das mensagens que seriam impressas no terminal. Logo a Aplicação Cliente precisa imprimir as informações em formato JSON, seguindo o padrão apresentado na Seção 3.1.1, para que esta possa ser interpretada pelo *cluster*.

O Executor sabe reconhecer o *status* de saída da Aplicação Cliente, sucesso ou erro, com base na análise do código de saída da aplicação. Isso é importante porque é comum que mensagens de erro e de alerta (*warnings*), sejam enviadas à saída de erro (STDERR). No entanto, as ações para tratamento de exceção por parte do *cluster* serão tomadas apenas se o código de saída indicar um erro. Logo, a Aplicação Cliente pode ter sido bem sucedida e mesmo assim apresentar mensagens no canal de erro, que serão registradas em arquivo texto (*log*) para análise do usuário. Não é necessário utilizar um formato específico de texto para mensagens de erro, uma vez que essas informações serão apenas exibidas para o usuário.

O Linux da suporte a linguagens compiladas e interpretadas. Códigos compilados geram arquivos que são executados diretamente pelo sistema operacional. As linguagens interpretadas utilizam interpretadores para executar suas aplicações. Nesse caso é necessário informar ao sistema operacional qual interpretador ele deve utilizar para executar o arquivo. Isso pode ser feito diretamente no arquivo do programa utilizando-se do *Shebang "#!"*. Essa notação é adicionada a primeira linha do arquivo seguida do caminho do interpretador que deve ser utilizado. Por exemplo, a instrução "#!/usr/bin/octave" indica que se utilizará o octave como interpretador. O sistema operacional utiliza isso para decidir qual interpretador será utilizado.

Outro detalhe ao ser observado ao executar uma aplicação de terminal é o diretório de trabalho. Normalmente esse é o próprio diretório em que a aplicação se encontra. Para evitar falhas quando uma aplicação tenta carregar outro arquivo (outro *script*, por exemplo), via caminhos relativos de diretório (../diretorio, por exemplo), optou-se por definir estaticamente qual será o diretório de trabalho. Isso é realizado por meio de um parâmetro de inicialização do *cluster* chamado JOBS_WORKDIR. Dessa forma a Aplicação Cliente pode fazer referência a outros arquivos considerando o caminho definido nesse parâmetro como diretório raiz.

Verifica-se que para ocorrer a comunicação entre *cluster* e Aplicação Cliente é necessário que esta seja capaz de: ler e escrever de/para um terminal; identificar parâmetros e extrair valores numa estrutura idêntica à ARGV; e manipular dados em formato JSON. Essas funcionalidades estão presentes na maioria das linguagens de programação, tornando o *cluster* flexível. Para facilitar a utilização, criou-se uma biblioteca que já implementa essas funcionalidades na linguagem Octave. Essa pode ser usada como exemplo para codificação em outras linguagens.

3.1.7 Configuração do servidor DRBL e inicialização via rede

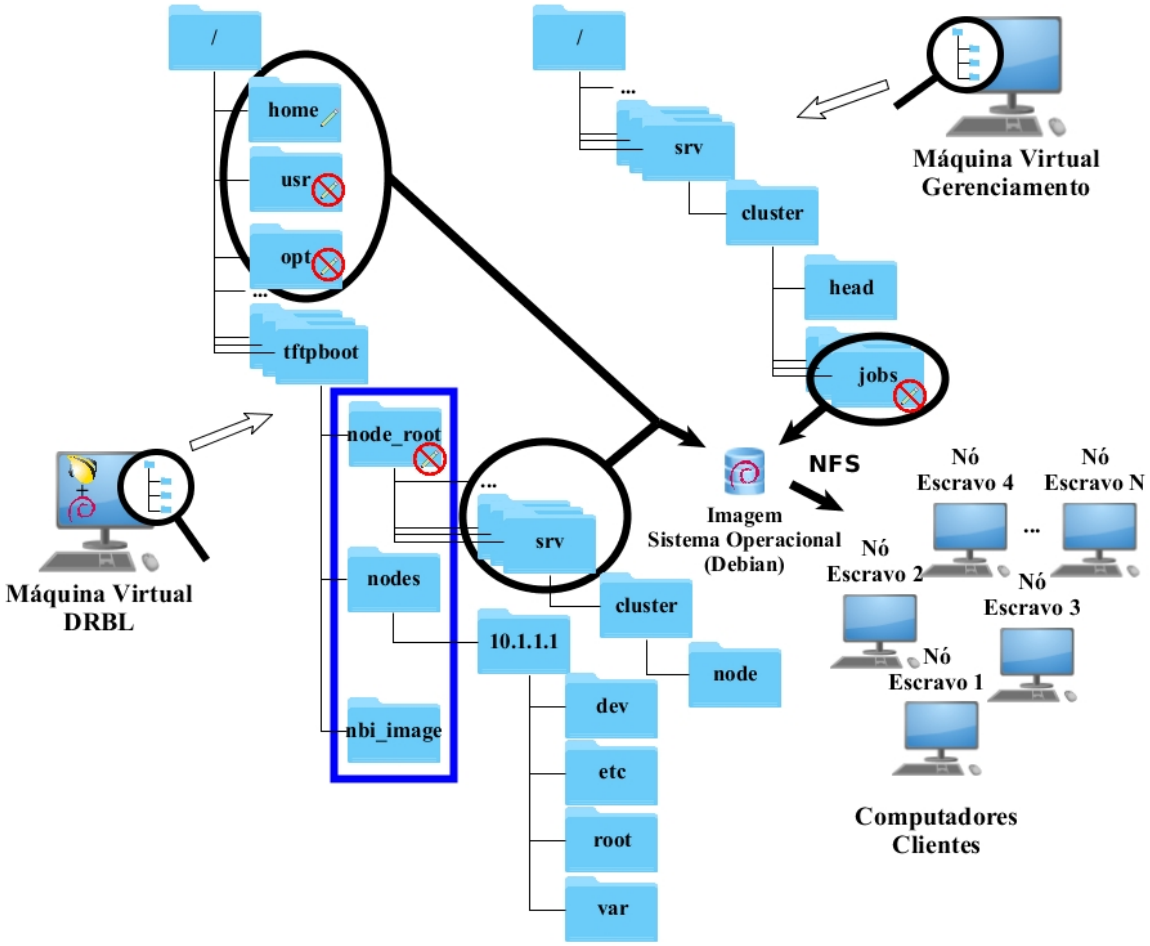
DRBL é um *software* que permite disponibilizar um sistema operacional Linux através da rede para múltiplos computadores simultaneamente. Diferentemente dos "terminais burros" (*thin clients*), o DRBL concede grande acesso aos recursos locais dos equipamentos, como processador e memória RAM. Dessa forma, o sistema operacional pode usufruir do potencial computacional do equipamento sem a necessidade de instalação.

O DRBL ainda possui utilitários que tornam possível incluir pacotes e arquivos à imagem do sistema operacional disponibilizado pela rede. Assim, é possível instalar e configurar

as dependências de pacotes de uma aplicação no servidor DRBL para criar um ambiente de *software* homogêneo entre vários computadores.

Para possibilitar o carregamento de um sistema operacional, o DRBL faz uma cópia da estrutura dos diretórios do sistema em que foi instalado. Isso é feito copiando a estrutura de diretórios do sistema para */tftpboot/node_root*. Esse diretório é exportado via NFS e mapeado para a raiz do sistema de arquivos dos computadores clientes, conforme apresentado na Figura 16.

Figura 16: Diretórios do sistema operacional da VM DRBL e VM de Gerenciamento envolvidos na geração da imagem do sistema operacional dos Nós Escravos.



Fonte: Próprio autor.

Um sistema operacional mínimo, contido no diretório */tftpboot/nbi_image*, é carregado nos clientes antes do sistema operacional principal. Esse possui a função de montar a estrutura de arquivos remota, tornando possível iniciar o sistema principal (Debian).

As pastas */home, /usr e /opt* são compartilhadas diretamente entre o servidor DRBL e os computadores clientes. Assim, modificações feitas pela VM DRBL nesses diretórios refletem de imediato nos clientes sem a necessidade de reiniciá-los. Por exemplo, quando se instala um pacote, seu executável fica em */usr/bin/*. Logo, esse estará disponível para os clientes após a

instalação. No entanto, os clientes podem alterar apenas o conteúdo em */home*. Modificações nesse diretório serão visíveis para a VM DRBL e demais clientes.

A pasta */tftpboot/nodes/* contém configurações personalizadas para cada cliente. Como se utilizou uma única imagem para todos os clientes, essa contém somente um diretório (*10.1.1.1*). O DRBL utiliza o primeiro IP (*Internet Protocol*) da faixa de endereços de rede dos clientes para nomeá-lo. Os serviços inicializados nos clientes são definidos no subdiretório *etc* desse diretório.

Os serviços criados na instalação de pacotes não são colocados na inicialização dos clientes automaticamente. Isso pode ser feito utilizando-se o comando *drbl-client-service (nome serviço) on*. Para que o serviço inicie no cliente imediatamente é necessário utilizar o comando *drbl-doit -b -n -u root "nome serviço" start* ou senão reiniciá-lo.

As configurações comuns entre os clientes ficam na pasta */tftpboot/node_root*. Os executáveis e arquivos de configuração da Aplicação Agente foram inseridos no subdiretório *srv/cluster/node/* de *node_root*. Dessa forma essa aplicação estará no diretório */srv/cluster/node* nos computadores clientes.

Um serviço foi criado para que a Aplicação Agente inicie junto ao sistema dos computadores clientes. Para isso, criou-se um *script* de inicialização baseado no formato *System V init* (WARD, 2014) na pasta da Aplicação Agente, chamado *slave-node-service*. Um *link* simbólico que aponta para */srv/cluster/node/slave-node-service* foi incluído manualmente no diretório */tftpboot/nodes/10.1.1.1/etc/init.d/*. Assim, no cliente ele se encontrará em */etc/init.d/*. O comando *insserv -p /tftpboot/nodes/10.1.1.1/etc/init.d/ slave-node-service* foi utilizado para gerenciar os níveis de execução do serviço.

Para que os Nós Escravos tivessem acesso às tarefas da Aplicação Cliente, adicionou-se um ponto de montagem personalizado à imagem do sistema. A pasta contida na Aplicação de Gerenciamento, */srv/cluster/jobs*, foi exportada via NFS em modo *somente leitura*. Então editou-se o arquivo */etc/drbl/client-append-fstab* com as configurações apropriadas para incluir esse diretório na imagem dos clientes. Dessa forma, modificações feitas pela Aplicação de Gerenciamento no diretório de tarefas da Aplicação Cliente são refletidas em todos o Nós Escravos instantaneamente.

3.1.8 Métricas e Testes de desempenho

O TOP500 Team (2017) utiliza o relatório produzido pela suíte de *benchmark* Linpack (DONGARRA; LUSZCZEK; PETITET, 2003) para criar a lista dos 500 supercomputadores mais poderosos do mundo. Essa suíte submete o equipamento a uma série de cálculos complexos e então utiliza o número de operações de ponto flutuante realizadas por segundo (FLOP/s) para avaliar o desempenho computacional.

Essa ferramenta avalia o equipamento (parte física) utilizando seu próprio *software* para programação distribuída. Logo, não faria sentido testar o *cluster* com essa ferramenta, visto que deseja-se testar a eficiência do *software* de gerenciamento de operações distribuídas (parte lógica), não o equipamento.

Dessa forma, focou-se em verificar o aumento de desempenho percebido pelo usuário. Para isso mediu-se quantas vezes a aplicação se tornou mais rápida quando submetida ao *cluster* em comparação a execução em um único computador (*speedup*), dado pela Equação 3.1 (AMDAHL, 1967):

$$S(p) = \frac{T(1)}{T(p)}, \tag{3.1}$$

onde, $S(p)$ é o *speedup*, $T(1)$ o tempo de processamento sequencial e $T(p)$ é o tempo de processamento paralelo para p processadores.

Segundo Amdahl (1967), o *speedup* de uma aplicação é limitado pelo percentual de código inerentemente sequencial, tornando difícil uma paralelização perfeita. Além disso, aplicações distribuídas também são afetadas pela latência da comunicação e pelo desbalanceamento da carga de trabalho, que pode ocasionar momentos de ociosidade em alguns processadores. Por isso, dificilmente se obtém um ganho de desempenho igual ao número de processadores utilizados (*Speedup* Ideal), uma vez que sempre haverá algum tipo de perda (*overhead*) (EIJKHOUT; CHOW; GEIJIN, 2016). Para se verificar o percentual de aproveitamento dos recursos computacionais, calculou-se a eficiência do *cluster*, dada pela Equação 3.2 (EIJKHOUT; CHOW; GEIJIN, 2016):

$$E(p) = \frac{S(p)}{p}, \tag{3.2}$$

onde, $E(p)$ é a eficiência e $S(p)$ o *speedup* para p processadores. Essa é dada pela razão entre o *speedup* alcançado e o ideal esperado para a quantidade de processadores utilizados.

3.2 Materiais

No ambiente de desenvolvimento, testes e produção utilizou-se os mesmos modelos de computador, variando-se o número de equipamentos empregados simultaneamente. Respectivamente 4 máquinas como Nós Escravos do *cluster* no ambiente de testes e 60 no ambiente de produção, onde utilizou-se um laboratório de informática. Esses ambientes compartilham o mesmo servidor de gerenciamento do *cluster*. Os computadores utilizados são do modelo Dell Optiplex 9020, equipados com processadores Intel Core i7-4770 (*quad core*), 8Gb de memória RAM e placas de rede *gigabit*. A rede do ambiente de testes e produção utilizam *switchs* Enterasys Gibabit modelo B3G124-24, BRG124-48 e C3G124-48, os quais são interligados por fibra óptica.

Um único computador foi utilizado para desenvolvimento. Essa máquina possui 16 Gb de memória RAM que foram divididos entre clientes virtuais. Nesse equipamento, utilizou-se o sistema operacional Linux Ubuntu 16.04.02. O sistema de gerenciamento do *cluster* foi desenvolvido na linguagem Ruby, utilizando-se a implementação oficial escrita em C, na versão 2.4.1 (MATSUMOTO, 2017; COOPER, 2016). Para a instalação e gerenciamento das versões desta linguagem, utilizou-se a ferramenta Rbenv (2017). Também foram utilizados os módulos *Open3* (CARLSON; RICHARDSON, 2015) e *DRb* (SEKI, 2012). O primeiro é utilizado para executar outra aplicação e ter acesso às suas interfaces de comunicação *STDIN*, *STDOUT* e

STDERR para entrada e saída de dados. O segundo módulo é utilizado para chamadas de método remoto, RMI, utilizando um protocolo próprio da linguagem.

Além da vasta biblioteca de funcionalidades distribuídas juntamente com a linguagem, Ruby também possui um mecanismo próprio de empacotamento e distribuição de código chamado *Gem* (Rubygems, 2017). Isto permite que desenvolvedores empacotem novas funcionalidades e as distribuam para a comunidade. Neste projeto foram utilizadas as *gems*: *net-ssh* (BUCK; MANDELBAUM; FAZEKAS, 2017), *net-scp* (BUCK; MANDELBAUM, 2017) e *net-ping* (CHERNESKY, 2016) para incluir código atualizado no servidores (*deploy*) e testar se estão respondendo aos comandos; *daemons* (UEHLINGER, 2016) para transformar programas que rodam em primeiro plano ininterruptamente em serviços que respondam aos comandos de gerenciamento de serviço do Linux; *rspec* (BAKER; CHELIMSKY; MARSTON, 2017) para testes automatizados; *msgpack* (FURUHASHI; HULTBERG; TAGOMORI, 2017) e *msgpack-rpc* (FURUHASHI; KASHIHARA, 2017) para serialização/desserialização no formato *msgpack* e comunicação via interface *Unix Domain Socket*; *sys-cpu* (BERGER, 2015) para obter informações sobre a configuração do processador, como número de núcleos e a associação entre núcleos lógicos e físicos; *ffi* (MEISSNER, 2017) para utilizar trechos da API Linux escrita em C diretamente de um código Ruby, utilizada para definir a afinidade entre processos e núcleos; e *Oj* (OHLER, 2017) para serialização/desserialização no formato JSON, visto que esta é mais eficiente que o mecanismo padrão da linguagem para esse fim. Como ferramenta de ajuda à codificação e depuração utilizou-se a *IDE Rubymine* da empresa JetBrains (2017). A ferramenta *Git* (CHACON; STRAUB, 2014) e o serviço de hospedagem *Bitbucket* (ATLASSIAN, 2017) foram utilizados para versionamento do código.

No ambiente de testes e produção escolheu-se utilizar a distribuição Debian Jessie 8.x (MURDOCK, 1993; HERTZOG; MAS, 2015) para ser o sistema operacional do *host* e *guests* devido a sua estabilidade e baixo consumo de recursos. Para a virtualização utilizou-se KVM e QEMU (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016) com *drivers Virtio*, o que permite bom desempenho para *guests* Linux, visto que a comunicação acontece diretamente através do *kernel* do *host*. Para gerenciamento das máquinas virtuais utilizou-se o *software Virt Manager* (LACROIX, 2016). Para executá-lo diretamente a partir do *host*, instalou-se a interface gráfica Openbox (2016) com suas dependências mínimas. No entanto, para fins de economia de memória, a interface gráfica não inicia junto ao sistema, sendo necessário executar o comando *startx* para iniciá-la.

Para disponibilizar o sistema operacional dos Nós Escravos se utilizou o *software* DRBL. Esse foi instalado em um sistema operacional Debian Jessie 8.x. Para que o cliente possa iniciar um sistema operacional pela rede é necessário que a placa suporte o protocolo PXE (*Preboot eXECUTION Environment*) (Intel Corporation; SYSTEMSOFT, 1999) e que este modo possa ser habilitado na BIOS. Ambas as funcionalidades estão disponíveis nos computadores utilizados neste trabalho.

As aplicações clientes que executam sobre o *cluster* (*jobs*) e as bibliotecas utilitárias foram desenvolvidas na linguagem Octave versão 4.2 (EATON; BATEMAN; HAUBERG, 1997). A partir dessa versão a linguagem ganhou suporte ao paradigma de orientação à objetos, utilizado na construção da biblioteca de utilitários para comunicação com o *cluster*. Tanto *jobs* quanto essa biblioteca necessitam manipular texto em formato JSON, convertendo-o em estrutura de dados e vice-versa, para isto, utilizou-se o pacote *JSONlab* (FANG, 2017), compatível com Octave e Matlab.

4 EXPERIMENTOS E RESULTADOS

Ferramentas de inteligência computacional são muito utilizadas para resolver problemas em engenharia (HORTA; CASTRO; BRAGA, 2015). Em geral, esses problemas consomem muito tempo de processamento, pois possuem muitos parâmetros que precisam ser otimizados. Além disso, para se testar o desempenho desse tipo de aplicação, os experimentos precisam ser repetidos muitas vezes. Dessa forma, esses problemas são fortes candidatos a utilizarem o *cluster* RedBlue. Assim, desenvolveu-se duas aplicações de inteligência computacional para avaliar o desempenho do *cluster*, ambas visando a investigar o impacto da variação de parâmetros na solução de um determinado problema.

A primeira foi baseada no trabalho de Vasconcelos *et al.* (2001), que buscava descobrir que tipos de operadores e procedimentos eram mais apropriados para determinados tipos de algoritmos genéticos. Nesse, utilizava-se as funções analíticas *rastringin, peaks* e *degree* sobre um domínio restrito para gerar um espaço de busca. Contava-se o número de sucessos dos algoritmos para diferentes parâmetros, ou seja, quantas vezes esses convergiram para o mínimo global. Parte desse trabalho foi implementada, atendo-se ao SGA (*Simple Genetic Algorithm*). Inicialmente, manteve-se a mesma metodologia de testes do trabalho base, partindo-se de uma configuração fixa para depois utilizar os melhores operadores/procedimentos da iteração anterior na próxima iteração. Isso resultou em 17 configurações possíveis e um tempo médio de execução de 46 minutos para um único processador. Nomeou-se essa aplicação como GA-01. Modificou-se o algoritmo para que fossem testadas todas as possibilidades de combinações, resultando em 432 configurações e um tempo médio de execução de 25,63 horas (1537,8 minutos) para um único processador. Nomeou-se essa versão como GA-02.

A segunda aplicação usou uma rede neural *Multilayer Perceptron* (MLP) com retropropagação (RUSSEL; NORVIG, 2013) para reconhecimento de caracteres. Nessa, buscava-se verificar o impacto do número de neurônios e camadas ocultas na acurácia das classificações obtidas. Para isso, utilizou-se o pacote *nnet* (SCHMID, 2010), que implementa apenas o algoritmo de treinamento Levenberg-Marquartdt (YU; WILAMOWSKI, 2011).

O conjunto de dados (*dataset*) *Letter Image Recognition Data* (SLATE, 1991) foi escolhido para testar a aplicação. O objetivo desse *dataset* é identificar qual letra maiúscula (A até Z) é formada a partir de um conjunto de pixeis. Cada letra possui 16 atributos numéricos que representam a distribuição dos pixeis na figura. Esse conjunto é formado por 20000 padrões únicos, criados a partir da distorção de 20 fontes de caracteres.

Modelou-se a rede neural com 16 neurônios na camada de entrada, uma para cada atributo, e 26 neurônio na de saída, um para cada letra possível. Variou-se o número de camadas ocultas de 1 a 2, com número de neurônios de 1 a 28 e 0 a 28, respectivamente. Utilizou-se a função de ativação *logsig* para as camadas ocultas e de saída. Como critério de parada, utilizou-se um máximo de 50 épocas e uma função objetivo com erro quadrático médio menor que 0,001. Apenas os dados de entrada foram normalizados, passando a ter média zero e desvio padrão um. Para a fase de treinamento reservou-se 20% (4000 padrões) do *dataset*, 10% (2000 padrões) para

validação e o restante para o teste de classificação (14000 padrões). Nos demais parâmetros da rede neural utilizou-se os valores padrão do pacote *nnet*. Logo, ao variar o número de camadas e de neurônios obteve-se 812 configurações possíveis. No entanto, destaca-se que o pacote *nnet* utiliza todos os núcleos disponíveis por padrão. Logo, considerou-se o tempo de execução ($T(1)$) em um computador (4 núcleos) e não em um único processador para o cálculo do *speedup*, totalizando cerca de 28 horas (1680 minutos). Portanto, $T(p)$ foi considerado como tempo de execução para p máquinas de 4 núcleos. A aplicação de rede neural foi nomeada como NN-01.

Todas as aplicações desenvolvidas foram dividias em três tarefas com responsabilidades bem definidas: configuração, processamento e agregação de resultados. A de configuração é responsável por criar as tarefas de processamento, distribuindo os parâmetros entre essas. A de processamento contém as regras de negócio para o problema a ser resolvido e se comporta de diferentes maneiras dependendo do conjunto de parâmetros. A tarefa de agregação de resultados é responsável por consolidar e formatar a saída das tarefas de processamento, exibindo informações úteis para o usuário.

Apenas as tarefas de processamento executam em paralelo. Porém, o tempo de execução dessas instâncias é elevado o suficiente para que se despreze o tempo gasto pelas tarefas de configuração e agregação de resultados. Assim, para simplificar a análise, considerou-se que as Aplicações Clientes desenvolvidas são altamente paralelizáveis. Logo, as perdas de desempenho serão atribuídas as operações administrativas do *cluster*, tornando possível medir o tempo perdido nessas operações.

A aplicação GA-01 foi executada apenas no ambiente de testes do *cluster* (Seção 3.2), com 4 Nós Escravos. Cada tarefa executou a aplicação GA-01 sequencialmente. Isso é, todas as instâncias executaram cópias idênticas da versão sequencial da aplicação GA-01. Testou-se com 4, 8, 12 e 16 tarefas de processamento distribuídas igualmente entre as mesmas quantidades de *Workers*. Ou seja, com 4 tarefas, utilizou-se 1 *Worker* em cada computador, com 8 tarefas, 2 *Workers* em cada computador, e assim sucessivamente. Cada configuração foi testada em dois cenários. O primeiro utilizando o escalonador de processos padrão do sistema operacional e o segundo utilizando afinidade entre processos e núcleos de processamento. Este mecanismo foi apresentado na Seção 3.1.5. Nomeou-se esse experimento como EXP-01.

A aplicação GA-02 foi executada no ambiente de produção (Seção 3.2). Cada máquina executou 4 tarefas simultaneamente, isto é, cada Nó Escravo instanciou 4 *Workers*. As simulações foram executadas com a opção CPU_AFFINITY ativada. Para testar a escalabilidade do *cluster* se variou o número de Nós Escravos, mas manteve-se o número de tarefas inalterado. Logo, 432 tarefas de processamento foram executadas em um *cluster* composto por 10, 20, 30, 40, 50 e 60 computadores. Nomeou-se esse experimento como EXP-02.

A aplicação NN-01 foi executada em *clusters* com os mesmos tamanhos utilizados por GA-02, de 10 à 60. No entanto essa aplicação possui 812 tarefas de processamento. Cada computador recebeu uma única tarefa de cada vez devido ao alto consumo de memória da aplicação. Apesar disso, todos os processadores disponíveis foram utilizados, uma vez que

a implementação de rede neural possui esta capacidade. Por essa razão as simulações foram executadas com a opção CPU_AFFINITY desativada. Nomeou-se esse experimento como EXP-03.

Em todos os experimentos se calculou o *speedup* e a eficiência do *cluster* para cada configuração testada. A Tabela 1 resume as configurações utilizadas nos experimentos e o tempo de execução sequencial utilizado para o cálculo das métricas.

Tabela 1: Resumo das configurações testadas para cada experimento no *cluster*

Experimento	**Aplicação**	**Tempo sequencial (minutos)**	**Número de máquinas**	**Número de tarefas**	***Workers* por máquina**	**CPU *affinity***
EXP-01	GA-01	184, 368, 552 e 736[1]	4	4, 8, 12 e 16	1, 2, 3 e 4	sim e não
EXP-02	GA-02	1537,8	10, 20, 30, 40, 50 e 60	432	4	sim
EXP-03	NN-01	1680	10, 20, 30, 40, 50 e 60	812	1	não

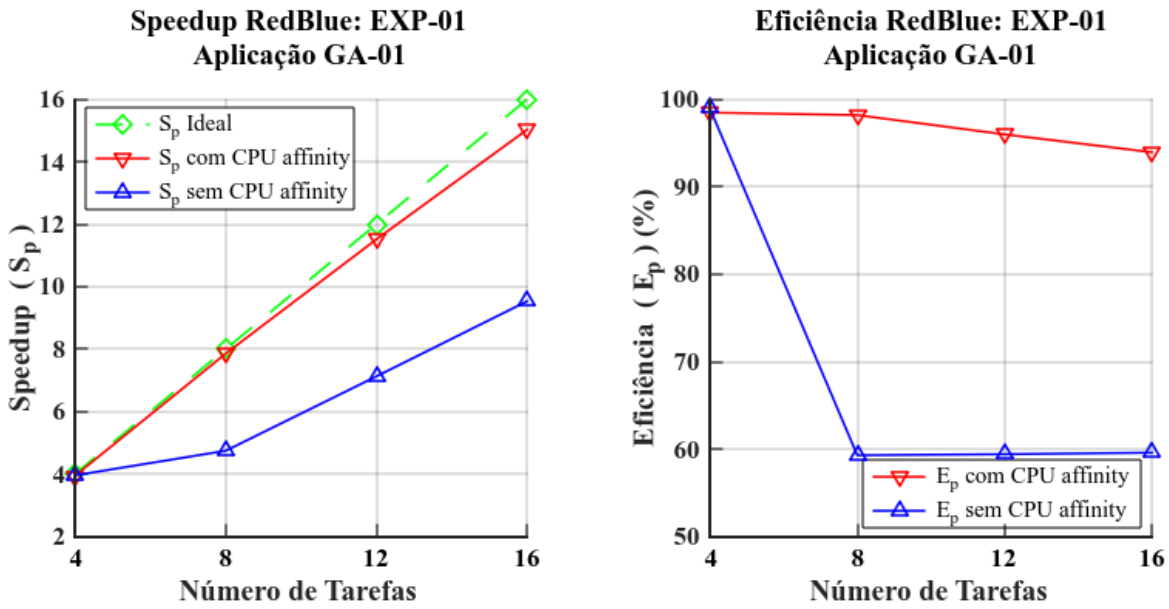
Fonte: Próprio autor.

[1]Na simulação EXP-1 se executou em paralelo um variado número de instâncias idênticas da aplicação GA-01, aumentando proporcionalmente o número de *Workers* utilizados. Logo, para esse experimento, calculou-se o tempo de execução sequencial como sendo o número de instâncias executadas em paralelo multiplicado pelo tempo médio gasto por uma instância de GA-01, 46 minutos.

A fim de facilitar a análise dos resultados, construiu-se os gráficos de *speedup* e eficiência de cada experimento. Para o gráfico de *speedup* se adicionou seu valor ideal para fim de comparação.

Na Figura 17 são apresentados os resultados obtidos para o experimento EXP-01. Nota-se um desempenho superior ao se utilizar afinidade entre processos e núcleos frente ao mecanismo padrão de alocação de processos do sistema operacional. O *speedup* para o cenário com $CPU\_AFFINITY$ ativo se aproximou do ideal enquanto que com o mecanismo padrão se manteve distante. Ao observar o gráfico de eficiência se verifica que essa diferença chegou a cerca 40% para 8 tarefas e a 35% para 16 tarefas (quatro tarefas por computador). Na abordagem com afinidade de processos houve uma perda de eficiência de apenas 5%, em relação ao *speedup* ideal, na execução de 16 tarefas. Para o mecanismo padrão houve queda brusca de eficiência na transição de 4 para 8 tarefas, estabilizando-se em cerca de 60% para 12 e 16 tarefas. Verificou-se que o desempenho ruim do mecanismo padrão se deve a alocação de duas tarefas a núcleos lógicos pertencentes ao mesmo núcleo físico do processador, causando disputa por recursos. Isso não ocorre ao se definir a afinidade das tarefas, uma vez que o *cluster* faz considerações

Figura 17: Gráficos do *speedup* e eficiência para experimento EXP-01.



(a) *Speedup* (b) Eficiência

Fonte: Próprio autor.

sobre a distribuição dos núcleos lógicos. No entanto, na implementação atual, uma tarefa fica restrita apenas a um núcleo, o que pode causar deterioração do desempenho em aplicações com várias *threads* ou processos. Isso porque nesse caso essas seriam alocadas em um único núcleo, enquanto os demais permaneceriam sub-utilizados.

Na Figura 18 são apresentados os resultados obtidos para o experimento EXP-02.

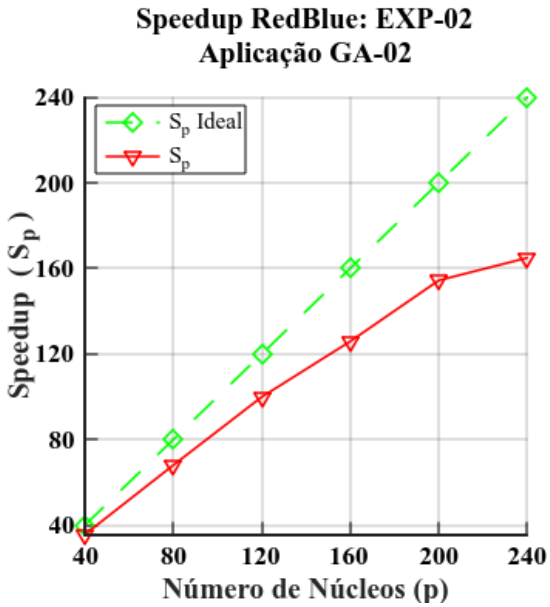
Observa-se uma diminuição gradual do *speedup* com redução mais acentuada entre 200 e 240 núcleos. Para 240 núcleos a eficiência foi de cerca de 68%. A queda mais expressiva de eficiência ocorreu no aumento de 200 para 240 núcleos, uma perda de cerca de 7%. Entre 80 e 120, e também entre 160 e 200 núcleos utilizados, houve uma perda de eficiência relativamente menor do a observada entre 40 e 80 núcleos, 120 e 160 núcleos, e 200 e 240 núcleos. Isso indica que entre 80 e 120, e também entre 160 e 200 núcleos, houve uma melhor distribuição de tarefas.

Na Figura 19 são apresentados os resultados obtidos para o experimento EXP-03.

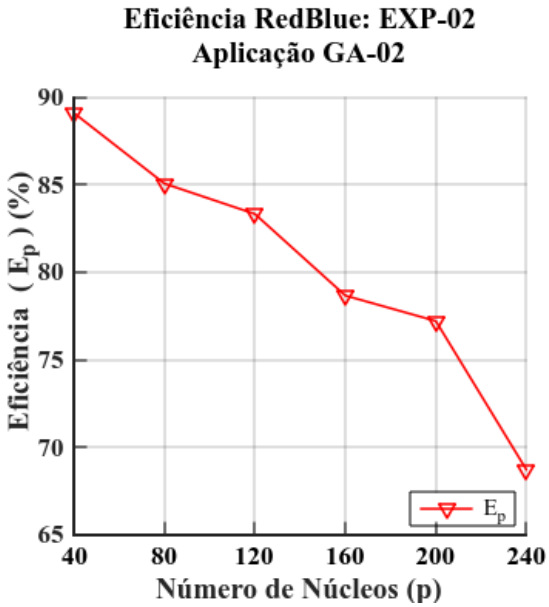
Observa-se que houve uma baixa perda de eficiência para todos os tamanhos de *cluster*. A maior perda ocorreu na transição entre 20 e 30 computadores, cerca de 2%. As demais perdas se mantiveram em cerca de 1% a cada 10 computadores adicionados. Para 60 máquinas houve um aproveitamento de 93%. O *speedup* manteve-se próximo ao ideal para todas as configurações testadas nesse experimento.

Embora as aplicações GA-02 e NN-01 tenham utilizado os mesmos tamanhos de *clusters*, nota-se uma diferença de desempenho expressiva. Uma das explicações para isso é o número de requisições simultâneas. Na Figura 20 se apresenta uma comparação entre máximo de requisições simultâneas possíveis de cada aplicação ao variar o tamanho do *cluster*. O gráfico

Figura 18: Gráficos do *speedup* e eficiência para experimento EXP-02.



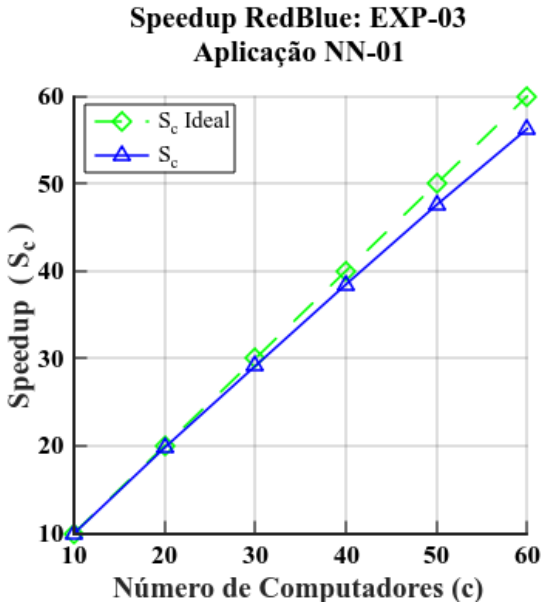
(a) *Speedup*



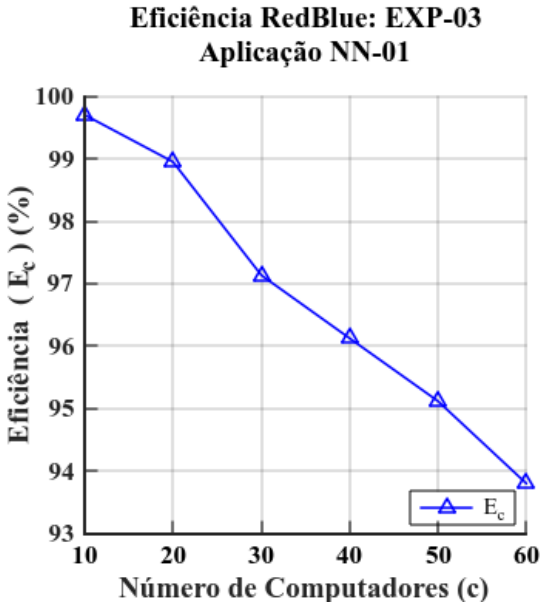
(b) Eficiência

Fonte: Próprio autor.

Figura 19: Gráficos do *speedup* e eficiência para experimento EXP-03.



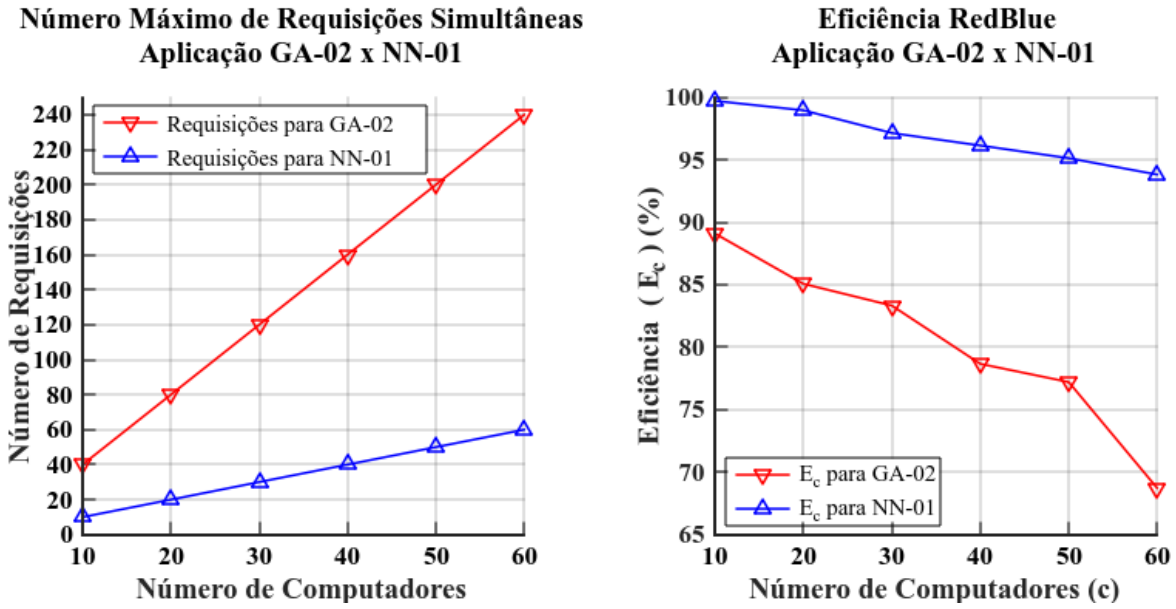
(a) *Speedup*



(b) Eficiência

Fonte: Próprio autor.

Figura 20: Gráficos do número máximo de requisições simultâneas possíveis e eficiência, uma comparação entre as aplicações GA-02 e NN-01.



(a) Número máximo de requisições possíveis

(b) Eficiência

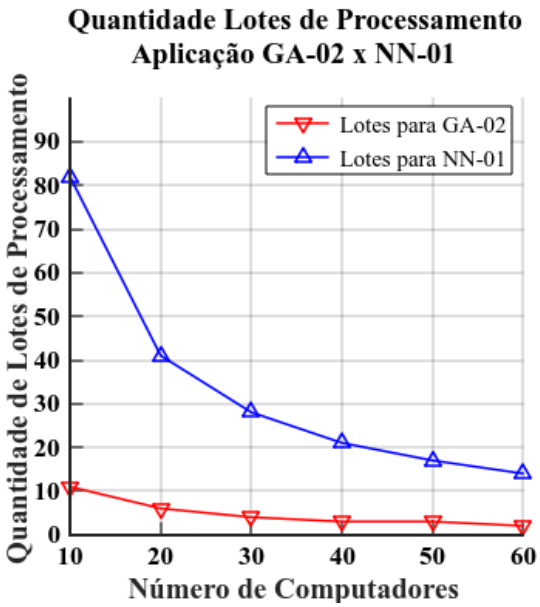
Fonte: Próprio autor.

da eficiência foi adicionado para se verificar a relação entre número de requisições e a eficiência obtida.

Observa-se que a diminuição da eficiência é proporcional ao aumento do número de requisições simultâneas. Isso indica que há um gargalo no Gestor de Requisições, fazendo com que atrasos adicionais apareçam a medida que o *cluster* aumenta de tamanho. De fato esta é uma característica de sistema distribuídos com gestão centralizada (TANENBAUM; STEEN, 2007). No entanto, essa pode ser amenizada ao se utilizar apenas requisições assíncronas e não bloqueantes. Ambas essas características ainda podem se aprimorados no RedBlue, que utiliza um sistema de sincronização de requisições que permite atender apenas uma por vez.

Um outro fator que pode causar deterioração do desempenho é o desbalanceamento de carga no lote final de processamento. O tamanho do lote é definido pelo número de tarefas que podem executar simultaneamente. Para as aplicações GA-02 e NN-01 esse limite é definido pelo número total de *Workers* disponíveis para cada tamanho do *cluster*. Isso acontece porque o número de tarefas de processamento é superior ao total de *Workers* para as configurações testadas. Logo, o número de lotes de processamento pode ser calculado se dividindo o total de tarefas pelo número total de *Workers*. Provavelmente o resultado dessa divisão não será exato, o que significa que será necessário adicionar mais um lote para se executar as tarefas remanescentes. Portanto, existirão processadores ociosos nesse último lote, o que acarretá na diminuição da eficiência. Na Figura 21 é apresentada uma comparação da quantidade de lotes necessários para aplicação GA-02 e NN-01 para os tamanhos de *clusters* testados.

Figura 21: Número de lotes de processamento.



Fonte: Próprio autor.

Observa-se que para aplicação GA-02 o aumento do *cluster*, de 30 a 50 computadores, possui pouco impacto na diminuição do número de lotes necessários. Enquanto o número de computadores cresceu em 66% nessa faixa, o número de lotes de processamento diminuiu em apenas 25%, constatando o desperdício de recursos.

Uma das formas de se atenuar esse problema é alocar mais tarefas que o número de processadores em lotes intermediários. Assim seria possível evitar a subutilização de recursos no último lote. Isso sacrificaria o desempenho de algumas tarefas, devido ao compartilhamento de recursos, mas poderia melhorar o desempenho geral. Outra ressalva seria a quantidade de memória disponível em cada computador, que poderia impedir que mais tarefas fossem adicionadas simultaneamente.

5 DISCUSSÃO

A plataforma apresentada neste trabalho para o desenvolvimento de simulações computacionais se mostrou viável para utilização em laboratórios de informática. Em especial para laboratórios que são utilizados para múltiplas finalidades, como em instituições de ensino, visto que é possível ativar o modo *cluster* apenas quando necessário. A capacidade do *cluster* se adaptar a quantidade de máquinas automaticamente e a necessidade de instalação em apenas um computador também são pontos positivos para utilizá-lo nesse tipo de ambiente. Outra característica interessante é a possibilidade de utilizar apenas as duas primeiras camadas da arquitetura. Dessa forma é possível ter um ambiente de desenvolvimento "*stand alone*" em um computador isolado. Isso é útil para que os usuários possam desenvolver e testar as aplicações em seus computadores pessoais. Ambas propriedades tornam a plataforma RedBlue interessante para instituições de ensino que não possuem uma plataforma dedicada de alto desempenho, mas têm laboratórios de informática disponíveis.

A plataforma também se mostrou capaz de acelerar as simulações computacionais. Os resultados dos experimentos mostraram uma redução expressiva no tempo de execução das simulações testadas. No experimento EXP-02 o tempo de execução sequencial foi de cerca de 25,63 horas. Ao utilizar o *cluster* com 60 computadores o tempo de execução para esse experimento foi reduzido para 10 minutos em média. Para o experimento EXP-03 o tempo de execução foi reduzido de 28 horas para cerca de 30 minutos para a mesma quantidade de máquinas no *cluster*. No entanto, em todos os testes foram observadas perdas de eficiência a medida em que mais máquinas são adicionadas ao *cluster*. Porém, esse comportamento era esperado, visto que é comumente observado em outros trabalhos (YANG; HSIEH; CHEN, 2008; DATTI; UMAR; GALADANCI, 2015; SETIAWAN; MURDYANTORO, 2016).

O tempo economizado com a redução no tempo de execução das simulações computacionais cria novas possibilidades de aplicação no ensino e pesquisa, além de permitir que se aproveite o tempo economizado em outras atividades. Por exemplo, os tempos de execução obtidos no *cluster* para as aplicações GA-02 e NN-01 tornam sua utilização viável durante as aulas. Seria possível expor conceitos e fomentar debates para depois confrontar os resultados esperados com os obtidos pela simulação. A apresentação dos resultados na mesma aula permitiria que os alunos tivessem acesso a uma aplicação prática imediatamente após aprenderem os conceitos teóricos.

Na pesquisa, a obtenção rápida de resultados poderia reduzir o esforço realizado com soluções inadequadas, como, por exemplo, testes que se mostrem ineficazes após muitas horas de simulação. Caso fosse verificado mais rapidamente que a solução não é adequada, poderia se utilizar esse tempo para buscar outra solução. Além disso, o *cluster* também poderia ser utilizado para testar simultaneamente várias configurações diferentes de uma simulação.

Cada um dos experimentos apresentados neste trabalho foi desenvolvido com uma finalidade específica. No experimento EXP-01 se observou que o escalonamento de processos tem forte impacto sobre o desempenho quando múltiplas instâncias estão em execução no mesmo

computador. Originalmente esse experimento foi desenhado para verificar a degradação no desempenho a medida que o número de tarefas aumentava enquanto os recursos computacionais permaneciam constantes. Por essa razão se executou várias instâncias de uma tarefa com os mesmos parâmetros. Pensava-se que, ao executar instâncias idênticas de uma aplicação em um computador com múltiplos núcleos, obteria-se um tempo de execução muito próximo em todas as instâncias. Logo, ao considerar o tempo de execução como uma constante, seria possível observar o *overhead* inserido pela paralelização de tarefas realizada pela plataforma. Porém, não foi o que aconteceu. Ao executar de duas a quatro instâncias no mesmo computador se verificou um aumento do tempo de execução da aplicação cliente, mesmo com memória RAM suficiente para isso. Esse problema foi identificado como sendo uma alocação inadequada de processos nos núcleos lógicos dos processadores causada pelo escalonador do sistema. Por essa razão se desenvolveu uma forma de alocação de processos otimizada para o *cluster*. Esse foi um achado importante, uma vez que permitiu utilizar todos os núcleos de uma máquina e ainda assim obter tempos de execução comparáveis a execução de uma instância única. Ao utilizar a afinidade entre processos (CPU_AFFINITY), o tempo de cada instância foi penalizado em apenas 4,7% em média. Para o mecanismo padrão o aumento do tempo de execução de cada instância chegou a 63,2% para quatro núcleos em comparação ao tempo de uma instância em um computador ocioso. Logo, verificou-se que o mecanismo de alocação de processos desenvolvido para o *cluster* é mais adequado a aplicações *single thread*. Para aplicações com múltiplas *threads* é preferível utilizar o mecanismo de alocação padrão do sistema operacional para evitar a concentração das *threads* em um único núcleo.

O experimento EXP-02 foi desenvolvido para mostrar que com ajuda de um *cluster* de computadores seria possível expandir o número de parâmetros testados e manter o tempo de execução baixo. Nesse experimento se executou a aplicação GA-02, que testou 432 combinações de parâmetros. Para 60 computadores essa gastou cerca de 10 minutos. A mesma foi desenvolvida a partir da aplicação GA-01, que testava apenas 17 variações de parâmetros, gastando cerca de 46 minutos. Ou seja, aumentou-se o número de combinações testadas em 25,4 vezes e ainda assim se reduziu o tempo de execução em 78%.

O experimento EXP-03 foi desenvolvido com o intuito de mostrar a aplicabilidade da plataforma a problemas muito sensíveis a variação de parâmetros, no caso, redes neurais artificiais. A calibração dos parâmetros em redes neurais é considerada uma tarefa complexa. Embora existam algumas estratégias para configurá-la, não há garantias que essas obterão bons resultados para todos os problemas. Dessa forma, é necessário modificar e testar os parâmetros várias vezes até encontrar uma boa configuração para o problema, uma tarefa que pode tomar muito tempo. Ao utilizar um *cluster* seria possível executar várias instâncias de uma rede neural com parâmetros diferentes simultaneamente e escolher aqueles que se mostrarem mais promissores. Dessa forma, reduziria-se o esforço e tempo gastos para encontrar a configuração ideal para o problema.

No experimento EXP-03 se obteve o melhor aproveitamento dos computadores disponíveis, cerca de 93% de eficiência, frente aos 67% de eficiência obtidos no experimento EXP-02 para 60 máquinas. Visto que ambos os experimentos foram executados com as mesmas quantidades de máquinas foi possível fazer comparações e especulações sobre a diferença obtida no critério eficiência. A primeira possibilidade é o número de requisições simultâneas. Para EXP-03 esse número poderia chegar a no máximo 60 enquanto que para EXP-02 poderia chegar a 240 requisições. Para 10 a 20 computadores no experimento EXP-02 poderiam ocorrer de 40 a 80 requisições simultâneas. Nesse intervalo a eficiência variou entre 89% a 85%, próximo a eficiência obtida para 60 computadores (60 requisições possíveis) em EXP-03, 93%. Logo, é possível concluir que a degradação no desempenho aumenta a medida que cresce o número de requisições simultâneas. A causa mais provável para isso é a incapacidade do Gestor de Requisições em tratar várias requisições simultaneamente. Isso provoca enfileiramento das requisições e consequentemente atrasos adicionais. Algumas soluções para esse problema seriam: utilizar apenas comunicação assíncrona; reduzir o tempo total de processamento das requisições otimizando o código; paralelizar os mecanismos da Aplicação de Gerenciamento; e utilizar múltiplas instâncias (processos) da Aplicação de Gerenciamento.

Outra explicação para a diferença da eficiência nos testes é causada pela estratégia de alocação de tarefas. O número de *Workers* é definido na inicialização do *cluster* e não é mais mudado. Dessa forma, tem-se um número máximo de tarefas que podem ser atendidas simultaneamente no *cluster*. Quando esse número de tarefas supera a quantidade de *Workers* é necessário que as demais tarefas esperem até que algum *Worker* esteja disponível. Em alguns casos o último lote terá poucas tarefas em execução, ou seja, a maioria das máquinas estará ociosa. Assim, a Aplicação de Gerenciamento irá esperar o término da última tarefa para retornar os resultados ao usuário. Observou-se esse comportamento nos experimentos realizados. Em EXP-03 houve um melhor aproveitamento dos lotes e comparação a EXP-02, o que explica a obtenção de uma eficiência melhor.

Algumas restrições para utilização da plataforma foram observadas. Apenas aplicações compatíveis com Linux funcionam no *cluster*. As aplicações utilizadas nos testes foram consideradas altamente paralelizáveis, visto que apenas as tarefas de configuração e coleta de resultados executaram sequencialmente. Logo, a utilização de aplicações menos paralelizáveis provavelmente resultará em piores tempos de execução. O conjunto de tarefas que executaram em paralelo tinha granularidade grossa. Isso significa que o tempo gasto para paralelizar as tarefas, iniciá-las e coletar os resultados é pequeno em relação ao tempo em que essas permanecem em execução. A utilização de tarefas com baixo tempo de execução traria deterioração drástica no desempenho do *cluster*. O *cluster* utiliza apenas troca de mensagens via rede como forma de comunicação, não há compartilhamento de memória. Isso implica que a quantidade de memória que uma tarefa pode utilizar está restrita a quantidade não utilizada no Nó Escravo em que essa está executando. A comunicação entre *cluster* e aplicação cliente acontece apenas no início e no fim da execução de uma tarefa. Logo não é possível que tarefas em execução comuniquem

entre si. Os resultados intermediários das tarefas são armazenadas em memória pela Aplicação de Gerenciamento, portanto está restrita a quantidade de memória disponível na Máquina Virtual de Gerenciamento.

A maior parte dessas limitações é necessária para simplificar o processo de desenvolvimento das aplicações por parte do usuário. Por exemplo, no trabalho de Datti, Umar e Galadanci (2015) se utilizou a interface MPI para programação distribuída. Logo era possível que sua aplicação se comunicasse com os demais nós do *cluster* a qualquer momento. Em contrapartida, a aplicação deveria lidar diretamente com questões referentes a distribuição e sincronização de tarefas. Setiawan e Murdyantoro (2016) apresenta um *cluster* construído com um sistema de imagem única, o Kerrighed. Esse sistema é capaz de operar simulando uma memória única entre os nós. No entanto utiliza *kernel* desatualizado para manter a compatibilidade com o Kerrighed e utiliza *scripts* adicionais para configurar a rede e detectar o número de nós do *cluster*. Ou seja, dificilmente será possível ter uma plataforma que traga um bom conjunto de utilidades sem algumas desvantagens associadas.

Apesar das limitações é possível concluir que a plataforma RedBlue atingiu seus objetivos. As estratégias para distribuição de tarefas e a ocultação da complexidade de um sistema distribuído foram dois dos principais achados deste trabalho. Logo, além da utilização da plataforma em laboratórios de informática, espera-se que as ideias propostas neste trabalho possam contribuir para o surgimento de novas aplicações voltadas a simulação computacional.

6 CONCLUSÃO

É indiscutível que o avanço tecnológico em diversas áreas do conhecimento só foi possível devido ao surgimento do computador. No entanto, ainda hoje, algumas dificuldades persistem devido a distância entre a computação como objeto de estudo e como ferramenta. Esse distanciamento ocorre porque apesar da computação ser aplicada a múltiplas áreas do conhecimento, poucos cursos de graduação incluem a ciência da computação com o peso necessário ao currículo.

Em virtude disso, muitos discentes, ou mesmo pesquisadores, deixam de utilizar ferramentas computacionais mais complexas por desconhecerem técnicas avançadas de programação. Um exemplo disso é a programação paralela e/ou distribuída, muito utilizada em supercomputadores ou *clusters*.

Para contornar esse quadro, esse trabalho propôs uma plataforma que permite utilizar os laboratórios de informática como *clusters* para desenvolvimento de simulações computacionais voltadas a pesquisa e ensino em engenharia. A plataforma permite abstrair conceitos complexos em computação paralela e distribuída, permitindo que pessoas com conhecimento básico em programação possam utilizá-la. Seu mecanismo de execução é flexível, tornando-a compatível com diferentes linguagens de programação. Além disso, a plataforma necessita ser instalada em apenas uma máquina.

Os resultados obtidos mostram uma redução de até 99% no tempo de execução do experimento no *cluster* com 60 máquinas, quando comparado a um único computador. Essa economia de tempo seria útil para pesquisas científicas. Atividades como refinamento de experimentos, desenvolvimento de novos testes ou análise mais profunda de resultados poderiam ser realizadas em menor tempo, aumentando a qualidade da pesquisa. Para o ensino, a redução do tempo de execução tornaria possível obter resultados de atividades práticas durante o tempo de aula. Além disso, tornaria possível praticar a modelagem de problemas e ter um retorno rápido sobre sua adequação ao problema proposto. Essas atividades colaboram para o surgimento de habilidades em modelagem e resolução de problemas, necessárias ao engenheiro moderno.

A plataforma foi desenvolvida para ser de fácil utilização. Dessa forma se espera que ela amplie o acesso de pessoas com diferentes níveis de conhecimento em programação às simulações computacionais. Assim, almeja-se tornar possível sua utilização como ferramenta para aquisição de conhecimento. A facilidade de uso poderia ser explorada no ensino, sendo que o potencial computacional dos laboratórios também poderia ser utilizado para execução de simulações científicas.

Dessa forma, considera-se que os objetivos propostos neste trabalho foram alcançados, visto que foi possível reduzir o tempo das simulações computacionais ao mesmo tempo que se simplificou o desenvolvimento das aplicações.

Como trabalhos futuros sugere-se:

1. Desenvolvimento de uma interface *web* como ferramenta de ajuda à codificação e depuração para o usuário, além de uma interface para gerenciamento e acompanhamento do estado dos computadores;
2. Testar a plataforma com alunos para descobrir se de fato as simplificações desenvolvidas para implementação de simulações computacionais são suficientes para sua utilização em atividades de aprendizagem;
3. Otimizar o código da plataforma para reduzir o tempo gasto em operações administrativas. Sugere-se a utilização de comunicação exclusivamente assíncrona, paralelização da Aplicação de Gerenciamento com utilização de múltiplos processos e controle dinâmico do número de *Workers* nos Nós Escravos;
4. Desenvolvimento de instaladores para várias distribuições Linux e criação de um repositório para possibilitar a distribuição de atualizações da plataforma. Além disso, documentar a plataforma e disponibilizar exemplos de utilização em diversas linguagens de programação;
5. Inclusão de um sistema de arquivos distribuídos para tornar possível utilizar o *cluster* com aplicações que consumam e produzam grandes quantidades de dados;
6. Desenvolver uma forma de compartilhar memória RAM entre os nós do *cluster* de forma transparente à aplicação cliente e assim permitir executar aplicações que consumam mais memória do que a disponível em um único computador.

REFERÊNCIAS

ALLEN, M. Introduction to molecular dynamics simulation. **Computational Soft Matter: From Synthetic Polymers to Proteins**, v. 23, n. 2, p. 1–28, 2004. ISSN 17412552.

AMDAHL, G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. **AFIPS Spring Joint Computer Conference, 1967. AFIPS '67 (Spring). Proceedings of the**, v. 30, p. 483–485, 1967. ISSN 18816096.

ATLASSIAN. **Bitbucket**. 2017. Disponível em: ⟨https://bitbucket.org⟩.

BACELLAR, H. Cluster: Computação de Alto Desempenho. **Ic.Unicamp.Br**, 2010.

BAKER, M.; BUYYA, R. Cluster computing: The commodity supercomputing. **Journal of Software - Practice & Experience**, v. 1, n. 1, 1999.

BAKER, S.; CHELIMSKY, D.; MARSTON, M. **Rspec gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/rspec⟩.

BERGER, D. J. **Sys-cpu gem**. 2015. Disponível em: ⟨https://rubygems.org/gems/sys-cpu⟩.

BOHN, B.; GARCKE, J.; IZA-TERAN, R.; PAPROTNY, A.; PEHERSTORFER, B.; SCHEPSMEIER, U.; THOLE, C. A. Analysis of car crash simulation data with nonlinear machine learning methods. **Procedia Computer Science**, Elsevier B.V., v. 18, p. 621–630, 2013. ISSN 18770509.

Bom dia Brasil. **Crise ameaça o Inpe e pode parar supermáquina da previsão do tempo**. 2017. Disponível em: ⟨http://g1.globo.com/bom-dia-brasil/noticia/2017/01/crise-ameaca-o-inpe-e-pode-parar-supermaquina-da-previsao-do-tempo.html⟩.

BROWN, R. **Engineering a Beowulf-style Compute Cluster**. Durham, NC: Lulu Press, 2007.

BUCK, J.; MANDELBAUM, D. **Net-scp gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/net-scp⟩.

BUCK, J.; MANDELBAUM, D.; FAZEKAS, M. **Net-ssh gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/net-ssh⟩.

BUYYA, R. **High performance cluster computing: Architectures and systems**. Upper SaddleRiver, NJ, USA: Prentice Hall PTR, 1999. v. 1. 999 p.

CAMBRIDGE. **Cambridge Centre for Teaching and Learning**. 2017. Disponível em: ⟨http://www.cctl.cam.ac.uk/⟩.

CARDOSO, S. O. d. O.; DICKMAN, A. G. Simulação computacional aliada à teoria da aprendizagem significativa: uma ferramenta para ensino e aprendizagem do efeito fotoelétrico. **Caderno Brasileiro de Ensino de Física**, v. 29, n. 2, p. 891–934, oct 2012. ISSN 2175-7941.

CARLSON, L.; RICHARDSON, L. **Ruby Cookbook: Recipes for object-oriented script**. 2. ed. Sebastopol: O Reilly, 2015. ISBN 9781449373719.

CHACON, S.; STRAUB, B. **Pro Git: Everything you need to know about gitt**. 2. ed. Califórnia, USA: Apress, 2014. ISBN 9781484200773.

CHEN, T.; CHEN, Y.; GUO, Q.; ZHOU, Z.-H.; LI, L.; XU, Z. Effective and efficient microprocessor design space exploration using unlabeled design configurations. **ACM Transactions on Intelligent Systems and Technology**, v. 5, n. 1, p. 1–18, dec 2013. ISSN 21576904.

CHERNESKY, C. **Net-ping gem**. 2016. Disponível em: ⟨https://rubygems.org/gems/net-ping⟩.

CHIRAMMAL, H. D.; MUKHEDKAR, P.; VETTATHU, A. **Mastering KVM virtualization**. Birmingham, UK: Packt Publishing Ltd, 2016. v. 53. 468 p. ISSN 1098-6596. ISBN 9781784399054.

COOPER, P. **Beginning Ruby**. 3. ed. Berkeley, CA: Apress, 2016. ISBN 978-1-4842-1279-0. Disponível em: ⟨http://link.springer.com/10.1007/978-1-4842-1278-3⟩.

COPELAND, D. B. **Build Awesome Command-Line Applications in Ruby: Control Your Computer, Simplify Your Life**. Dallas: The Pragmatic Programmers, 2013. 212 p. ISBN 97819377858.

COUTURE-BEIL, A. **rjson: JSON for R**. 2014. Disponível em: ⟨https://cran.r-project.org/web/packages/rjson/index.html⟩.

CROUCH, C. H.; MAZUR, E. Peer Instruction: Ten years of experience and results. **American Journal of Physics**, v. 69, n. 9, p. 970–977, sep 2001. ISSN 0002-9505.

DALGARNO, B.; KENNEDY, G.; BENNETT, S. The impact of students' exploration strategies on discovery learning using computer-based simulations. **Educational Media International**, v. 51, n. 4, p. 310–329, oct 2014. ISSN 0952-3987.

DASGUPTA, C. Investigating the use of Simulation Model for Teaching Engineering Design. 2016.

DATTI, A. A.; UMAR, H. A.; GALADANCI, J. A Beowulf Cluster for Teaching and Learning. **Procedia Computer Science**, v. 70, n. 70, p. 62–68, 2015. ISSN 18770509.

DONGARRA, J. J.; LUSZCZEK, P.; PETITET, A. The LINPACK Benchmark: past, present and future. **Concurrency and Computation: Practice and Experience**, v. 15, n. 9, p. 803–820, 2003. ISSN 1532-0626.

EATON, J. W.; BATEMAN, D.; HAUBERG, S. **Gnu octave**. GNU, 1997. Disponível em: ⟨http://folk.ntnu.no/joern/itgk/octave.pdf⟩.

ECKHARDT, M.; URHAHNE, D.; CONRAD, O.; HARMS, U. How effective is instructional support for learning with computer simulations? **Instructional Science**, Springer Netherlands, v. 41, n. 1, p. 105–124, jan 2013. ISSN 00204277.

ECMA-404. The JSON Data Interchange Format. **EMAC International**, v. 1st Editio, n. October, 2013. Disponível em: ⟨http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf⟩.

EIJKHOUT, V.; CHOW, E.; GEIJIN, R. van. **Introduction to High Performance Scientific Computing**. 2. ed. [S.l.]: Lulu.com, 2016. 554 p. ISBN 978-1-257-99254-6.

FANG, Q. **JSONlab: a toolbox to encode/decode JSON/UBJSON files in MATLAB/Octave**. 2017. Disponível em: ⟨http://iso2mesh.sourceforge.net/cgi-bin/index.cgi?jsonlab⟩.

FISCHBORN, M. **Computação de alto desempenho aplicada à análise de dispositivos eletromagnéticos**. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2006.

Folha de São Paulo. **Sem pagar conta de luz, laboratório no Rio restringe uso de supercomputador - 22/06/2016 - Ciência - Folha de S.Paulo**. 2016. Disponível em: ⟨http://www1.folha.uol.com.br/ciencia/2016/06/1784420-sem-pagar-conta-de-luz-laboratorio-no-rio-pode-desligar-supercomputador.shtml⟩.

FURUHASHI, S.; HULTBERG, T.; TAGOMORI, S. **Msgpack gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/msgpack⟩.

FURUHASHI, S.; KASHIHARA, S. **Msgpack-rpc gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/msgpack-rpc⟩.

GARLAND, J. Unravelling the complexity of signalling networks in cancer: A review of the increasing role for computational modelling. **Critical Reviews in Oncology/Hematology**, Elsevier Ireland Ltd, v. 117, p. 73–113, 2017. ISSN 10408428.

GHOLKAR, N.; MUELLER, F.; ROUNTREE, B. A Power-Aware Cost Model for HPC Procurement. In: **2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. Chicago, USA: IEEE, 2016. p. 1110–1113. ISBN 978-1-5090-3682-0.

GRECA, I. M.; SEOANE, E.; ARRIASSECQ, I. Epistemological Issues Concerning Computer Simulations in Science and Their Implications for Science Education. **Science and Education**, v. 23, n. 4, p. 897–921, 2014. ISSN 15731901.

HAIGH, T.; PRIESTLEY, M.; ROPE, C. Engineering 'The miracle of the ENIAC': Implementing the modern code paradigm. **IEEE Annals of the History of Computing**, v. 36, n. 2, p. 41–59, 2014. ISSN 10586180.

HAIGH, T.; PRIESTLEY, M.; ROPE, C. Los Alamos Bets on ENIAC: Nuclear Monte Carlo Simulations, 1947-1948. **IEEE Annals of the History of Computing**, v. 36, n. 3, p. 42–63, jul 2014. ISSN 1058-6180.

HAIGH, T.; PRIESTLEY, M.; ROPE, C. Reconsidering the stored-program concept. **IEEE Annals of the History of Computing**, v. 36, n. 1, p. 4–17, 2014. ISSN 10586180.

HALL, S.; WAITZ, I.; BRODEUR, D.; SODERHOLM, D.; NASR, R. Adoption of Active Learning in a Lecture-based Engineering Class. In: **32nd Annual Frontiers in Education**. Boston, USA: IEEE, 2002. v. 1, p. 9–15. ISBN 0-7803-7444-4. ISSN 0190-5848.

HARVARD. **Harvard: The Derek Bok Center for Teaching and Learning**. 2017. Disponível em: ⟨https://bokcenter.harvard.edu/⟩.

HENNESSY, J. L.; PATTERSON, D. a. **Computer Architecture, Fourth Edition: A Quantitative Approach**. 4. ed. [S.l.: s.n.], 2007. 704 p. ISSN 00262692. ISBN 0123704901.

HERTZOG, R.; MAS, R. **The Debian Administrator ' s Handbook**. [S.l.: s.n.], 2015. 496 p. ISBN 9791091414005.

HORTA, E. G.; CASTRO, C. L. de; BRAGA, A. P. Stream-Based Extreme Learning Machine Approach for Big Data Problems. **Mathematical Problems in Engineering**, v. 2015, p. 1–17, 2015. ISSN 1024-123X.

Intel Corporation; SYSTEMSOFT. **Preboot Execution Environment (PXE) Specification**. [S.l.], 1999. 103 p.

JANSSON, N.; HOFFMAN, J.; NAZAROV, M. Adaptive simulation of turbulent flow past a full car model. In: **State of the Practice Reports on - SC '11**. New York, New York, USA: ACM Press, 2011. v. 0, p. 1. ISBN 9781450311397.

JETBRAINS. **RubyMine**. 2017. Disponível em: ⟨https://www.jetbrains.com/ruby/⟩.

JONG, T. de. Technological advances in inquiry learning. **Science**, v. 312, n. 5773, p. 532–533, 2006. ISSN 00368075.

JONG, T. de; JOOLINGEN, W. R. V. Scientific Discovery Learning with Computer Simulations of Conceptual Domains. **Review of Educational Research**, Reigeluth & Schwartz Riley Smith, v. 68, n. 2, p. 179–201, jan 1998. ISSN 0034-6543.

KERRISK, M. **The Linux Programming Interface**. San Francisco, USA: No Starch Press, Inc., 2010. 1552 p. ISBN 978-1-59327-220-3.

KOLAR, R. L.; SABATINI, D. A. Coupling team learning and computer technology in project-driven undergraduate engineering education. **Frontiers in Education Fie'96 - 26th Annual Conference, Proceedings, Vols 1-3**, p. 172–175, 1996.

LACROIX, J. **Mastering Ubuntu Server**. Birmingham, UK: Packt Publishing, 2016. 430 p. ISBN 9789797942908.

LATTUCA, L. R.; TERENZINI, P. T.; VOLKWEIN, J. F. **Engineering Change: A Study of the Impact of EC2000**. Baltimore, USA, 2006. 30 p. Disponível em: ⟨www.ed.psu.edu/cshehttp://www.abet.org/wp-content/uploads/2015/04/EngineeringChange-executive-summary.pdf⟩.

LEPILLEUR, B. **JsonCPP**. 2010. Disponível em: ⟨https://github.com/open-source-parsers/jsoncpp⟩.

LIAO, Y.-k.; CHEN, Y.-w. The effect of Computer Simulation Instruction on student learning : A meta-analysis of studies in Taiwan. **Journal of Information Technology and Applications**, v. 2, n. 2, p. 69–79, 2007.

LOTTIAUX, R.; GALLARD, P.; VALLEE, G.; MORIN, C.; BOISSINOT, B. OpenMosix, OpenSSI and Kerrighed: a comparative study. In: **CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.** Lottiaux2015: IEEE, 2005. p. 1016–1023 Vol. 2. ISBN 0-7803-9074-1.

LOVE, R. **Linux System Programming: Talking Directly to the Kernel and C Library**. 2. ed. Sebastopol: O Reilly, 2013. 456 p. ISBN 9780596009588.

MARTIN, L. K.; SLATER, C. Should engineering education be more practical, more theoretical, or remain as it currently stands? **IEEE Potentials**, v. 31, n. 4, p. 6–8, jul 2012. ISSN 02786648.

MATSUMOTO, Y. **Ruby**. 2017. Disponível em: ⟨https://www.ruby-lang.org/⟩.

MATTHEW, N.; STONES, R. Beginning Linux ® Programming 4th Edition. p. 818, 2009.

MCHANEY, R. **Understandind Computer Simulation**. [S.l.]: bookboon.com, 2009. ISBN 978-87-7681-505-9.

MEISSNER, W. **Ffi gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/ffi⟩.

MIT. **MIT Teaching Learning Lab**. 2017. Disponível em: ⟨https://tll.mit.edu/help/guidelines-teaching⟩.

MONTEIRO, S. A. I.; RIBEIRO, R.; LEMES, S. d. S.; MUZZETI, L. R. **Educações na contemporaneidade: reflexão e pesquisa.** 1. ed. São Carlos - SP: Pedro & João Editores, 2011. ISBN 9788579930874.

MOURTZIS, D.; DOUKAS, M.; BERNIDAKI, D. Simulation in Manufacturing: Review and Challenges. **Procedia CIRP**, Elsevier B.V., v. 25, n. C, p. 213–229, 2014. ISSN 22128271.

MURDOCK, I. **Debian**. 1993. Disponível em: ⟨https://www.debian.org⟩.

NEGAHBAN, A.; SMITH, J. S. Simulation for manufacturing system design and operation: Literature review and analysis. **Journal of Manufacturing Systems**, v. 33, n. 2, p. 241–261, apr 2014. ISSN 02786125.

NEUBERT, J. C.; MAINERT, J.; KRETZSCHMAR, A.; GREIFF, S. The Assessment of 21st Century Skills in Industrial and Organizational Psychology: Complex and Collaborative Problem Solving. **Industrial and Organizational Psychology**, v. 8, n. 02, p. 238–268, jun 2015. ISSN 1754-9426.

NEUMANN, J. von. First Draft of a Report on the EDAVAC. **American Mathematical Society**, v. 15, n. 1, p. 1–10, 1945.

OHLER, P. **Oj gem**. 2017. Disponível em: ⟨https://rubygems.org/gems/oj⟩.

Openbox. 2016. Disponível em: ⟨https://wiki.debian.org/Openbox⟩.

PARKER, W. S. Simulation and Understanding in the Study of Weather and Climate. **Perspectives on Science**, v. 22, n. 3, p. 336–356, sep 2014. ISSN 1063-6145.

PINHEIRO, F. J. R. **Uma proposta de um sistema de imagem única para uso de computação em grade em organizações virtuais**. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2005.

PITANGA, M. **Computação em Cluster**. 2003. Disponível em: ⟨http://www.clubedohardware.com.br/artigos/computacao-em-cluster/153⟩.

Python Software Foundation. **JSON encoder and decoder**. 2017. Disponível em: ⟨http://docs.python.org/library/json.html⟩.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. 1. ed. Singapore: McGraw-Hill, 2004. ISBN 0072822562.

RAUBER, T.; RÜNGER, G. **Parallel Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-04817-3.

Rbenv. 2017. Disponível em: ⟨https://github.com/rbenv/rbenv⟩.

REAL, E. M.; FILHO, F. S. B. Uma proposta de um cluster de baixo investimento para o processamento paralelo de aplicações em computadores de uma mesma rede. In: **I Simpocomp - Simpósio da Computação, 2012**. Nova Andradina-MS: SIMPOCOMP, 2012.

ROBINSON, S. **Simulation : The Practice of Model Development and Use**. Chichester, England: John Wiley & Sons Ltd, 2004. ISBN 0-470-84772-7.

ROJAS, R.; HASHAGEN, U. **The First Computers: History and Architectures**. London: The MIT Press, 2001. v. 42. 615–617 p. ISSN 0021-1753. ISBN 0262181975.

ROSE, C. A. D.; NAVAUX, P. O. A. Fundamentos de Processamento de Alto Desempenho. In: **Anais da 2º Escola Regional de Alto Desempenho**. Pelotas: ERAD 2004, 2004. p. 3–29. ISBN 85-7669-055-1.

Rubygems. 2017. Disponível em: ⟨https://rubygems.org/⟩.

RUSSEL, S.; NORVIG, P. **Inteligência Artificial**. 3. ed. Rio de Janeiro: Campus, 2013. ISBN 978-85-352-3701-6.

RUTTEN, N.; JOOLINGEN, W. R. van; VEEN, J. T. van der. The learning effects of computer simulations in science education. **Computers & Education**, v. 58, n. 1, p. 136–153, jan 2012. ISSN 03601315.

SCHMID, M. **nnet**. 2010. Disponível em: ⟨https://octave.sourceforge.io/nnet/⟩.

SEKI, M. **The dRuby Book: Distributed and Parallel Computing with Ruby**. Dallas, USA: Pragmatic Programmers, 2012. 266 p. ISBN 9781934356937.

SETIAWAN, I.; MURDYANTORO, E. Commodity cluster using single system image based on Linux/Kerrighed for high-performance computing. In: **2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)**. Semarang, Indonésia: IEEE, 2016. p. 367–372. ISBN 978-1-5090-1434-7.

SHIAU, S.; HUANG, K.; SUN, C.; WANG, J.; TSAI, T.; NIFENECKER, J.-F.; CHEN, L.; ALAGAPPAN, N. **DRBL Project**. 2017. Disponível em: ⟨http://drbl.org/⟩.

SINGH, V. P. **System modeling and simulation**. New Delhi, India: New Age International Publishers, 2009.

SLATE, D. J. **Letter Recognition Data Set**. 1991. Disponível em: ⟨https://archive.ics.uci.edu/ml/datasets/letter+recognition⟩.

SOUNDARAJAN, N. Engineering Criteria 2000: the impact on engineering education. In: **FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference**. San Juan, Puerto Rico: IEEE, 1999. v. 1, p. 6. ISBN 0-7803-5643-8. ISSN 01905848.

STERN, F.; YANG, J.; WANG, Z.; SADAT-HOSSEINI, H.; MOUSAVIRAAD, M. Computational ship hydrodynamics: Nowadays and way forward. **International Shipbuilding Progress**, v. 60, n. 1-4, p. 3–105, 2013. ISSN 0020868X.

STEVENS, W. R. **UNIX Network Programming: Interprocess Comunications**. Upper Saddle River: Prentice Hall, 1999. 11–13 p. ISBN 0-13-081081-9.

STEVENS, W. R.; RAGO, S. A. **Advanced programming in the UNIX environment**. 3. ed. Upper Saddle River: Addison-Wesley, 2013. ISBN 978-0-321-63773-4.

TANENBAUM, A. s. **Computer Networks**. 4. ed. Amsterdam, Holanda: Prentice Hall Professional Technical Reference, 2002. 632 p. ISBN 0130661023.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms**. 2. ed. Upper Saddle River: Prentice Hall, 2007. 705 p. ISBN 0-13-239227-5.

TANG, F.; REN, A. Agent-Based Evacuation Model Incorporating Fire Scene and Building Geometry. **Tsinghua Science and Technology**, v. 13, n. 5, p. 708–714, oct 2008. ISSN 10070214.

TIWARI, A.; HOYOS, P. N.; HUTABARAT, W.; TURNER, C.; INCE, N.; GAN, X.-P.; PRAJAPAT, N. Survey on the use of computational optimisation in UK engineering companies. **CIRP Journal of Manufacturing Science and Technology**, v. 9, p. 57–68, may 2015. ISSN 17555817.

TOP500 Team. **TOP500: List Statistics**. 2017. Disponível em: ⟨https://www.top500.org/statistics/list/⟩.

UEHLINGER, T. **Daemons gem**. 2016. Disponível em: ⟨https://rubygems.org/gems/daemons⟩.

VALENTE, J. A. **Computadores e Conhecimento: repensando a educação**. 2. ed. Campinas-SP: UNICAMP/NIED, 1998. 501 p.

VASCONCELOS, J.; RAMIREZ, J.; TAKAHASHI, R.; SALDANHA, R. Improvements in genetic algorithms. **IEEE Transactions on Magnetics**, IEEE, v. 37, n. 5, p. 3414–3417, 2001. ISSN 00189464.

VELTEN, K. **Mathematical modeling and simulation: Introduction for Scientists and Engineers**. [S.l.]: Wiley-VCH, 2009. 365 p. ISSN 1361-648X. ISBN 978-3-527-40758-8.

WANG, F.; YIN, Z.; YAN, S.; ZHAN, J.; FRIZ, H.; LI, B.; XIE, W. Validation of Aerodynamic Simulation and Wind Tunnel Test of the New Buick Excelle GT. **SAE International Journal of Passenger Cars - Mechanical Systems**, v. 10, n. 1, p. 2017–01–1512, mar 2017. ISSN 1946-4002.

WARD, B. **How Linux Works: What every superuser should know**. [S.l.]: No Starch Press, 2014. 338 p.

WINSBERG, E. Computer Simulation in Science. In: ZALTA, E. N. (Ed.). **The Stanford Encyclopedia of Philosophy**. Summer 201. Metaphysics Research Lab, Stanford University, 2015. Disponível em: ⟨https://plato.stanford.edu/archives/sum2015/entries/simulations-science⟩.

YANG, C. T.; HSIEH, W. F.; CHEN, H. Y. Implementation of a diskless cluster computing environment in a computer Classroom. **Proceedings of the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC 2008**, p. 819–824, 2008.

YANG, X.-S.; KOZIEL, S.; LEIFSSON, L. Computational Optimization, Modelling and Simulation: Recent Trends and Challenges. **Procedia Computer Science**, v. 18, p. 855–860, 2013. ISSN 18770509.

YU, H.; WILAMOWSKI, B. Levenberg–Marquardt Training. In: **Industrial Electronics Handbook, vol. 5 – Intelligent Systems**. Boca Raton: CRC Press, 2011. cap. 12, p. 1–16. ISBN 978-1-4398-0283-0.